Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: May 17, 2018

S. Cheshire
Apple Inc.

November 13, 2017

Private Discovery Threat Considerations
draft-cheshire-dnssd-privacy-considerations-01

Abstract

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

Status of This Memo

Copyright Notice

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 be 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

[RFC5054] Taylor, D., Wu, T., Mavrogiannopoulos, N., and T. Perrin, "Using the Secure Remote Password (SRP) Protocol for TLS Authentication", RFC 5054, DOI 10.17487/RFC5054, November 2007, <https://www.rfc-editor.org/info/rfc5054>.

[RFC8235] Hao, F., Ed., "Schnorr Non-interactive Zero-Knowledge Proof", RFC 8235, DOI 10.17487/RFC8235, September 2017, <https://www.rfc-editor.org/info/rfc8235>.

[RFC8236] Hao, F., Ed., "J-PAKE: Password-Authenticated Key Exchange by Juggling", RFC 8236, DOI 10.17487/RFC8236, September 2017, <https://www.rfc-editor.org/info/rfc8236>.

Author’s Address

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

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

Internet Engineering Task Force S. Cheshire
Internet-Draft Apple Inc.
Intended status: Informational October 23, 2018
Expires: April 26, 2019

Service Discovery Road Map
draft-cheshire-dnssd-roadmap-03

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

Copyright Notice

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. They 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 communication 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 specification [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 some 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 a concrete 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. Depending on the specific use case, this step may be performed only rarely. For example, a user may do this just one once, the first time they configure their computer to use their preferred printer, and never again.

Once a desired printer has been chosen and configured, subsequent printing of documents is an operation performing the third step. This step may be done frequently, perhaps multiple times per day. This third step is important because, in a world of DHCP, IPv6 Stateless Autoconfiguration, and similar dynamic address allocation schemes, a printer's IP address could change from day to day, and to use the printer, its current address must be known. However, this third step need not be performed for every printer on the network, just the specific printer that is about to be used. Also, it is not necessary to repeat the second step again, learning the names of every printer on the network, if the client device already knows the name of the printer 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 should not only perform discovery at a single moment in time, but should also provide ongoing change notification (sometimes called "Publish & Subscribe"). Clients need to be notified in a timely fashion when new data of interest appears, when data of interest changes, and, equally importantly, when data of interest goes away ("goodbye packets"). 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 creation, update, and deletion, 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 determine the appropriate 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 broadly in Section 4 below. Client operation is then discussed in detail in Section 5, and server operation is discussed in detail in Section 6.

4. Service Discovery Namespace

When used with Multicast DNS [RFC6762] Service Discovery queries necessarily use the ".local" parent domain reserved for this purpose [SUDN].

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 advanced service discovery to be available even to people who haven't taken the step of registering and paying annually 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" [RFC8375]. Services published in this factory-default service discovery domain are not globally unique or globally resolvable, but they can 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] on the local link, queries are necessarily performed in the ".local" parent domain reserved for this purpose [SUDN].

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.

A decade of operational experience has shown that an asynchronous change notification protocol built on TCP is preferable for a variety of reasons, so the IETF is has developed 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 trivial 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] by hand, 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 technique in the direction of automatically populating the DNS namespace is the Service Discovery Proxy [DisProx]. This technology works 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 with a presence 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 communicating with 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 technique 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 to the basic DNS Update protocol 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 DHCP lease lifetime, and if the device fails to renew the DHCP 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 for address assignment, 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 to the basic DNS Update protocol is the introduction of information, carried using the EDNS(0) OWNER Option [Owner], that tells the Service Registration server that the device will be going to sleep to save power, and how the Service Registration server can wake it up again on demand when needed. 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.

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 some mesh networks.

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

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

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

[RFC6281] Cheshire, S., Zhu, Z., Wakikawa, R., and L. Zhang, "Understanding Apple's Back to My Mac (BTMM) Service", RFC 6281, DOI 10.17487/RFC6281, June 2011, <https://www.rfc-editor.org/info/rfc6281>.

[RFC6335] Cotton, M., Eggert, L., Touch, J., Westerlund, M., and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry", BCP 165, RFC 6335, DOI 10.17487/RFC6335, August 2011, <https://www.rfc-editor.org/info/rfc6335>.

[RFC6760] Cheshire, S. and M. Krochmal, "Requirements for a Protocol to Replace the AppleTalk Name Binding Protocol (NBP)", RFC 6760, DOI 10.17487/RFC6760, February 2013, <https://www.rfc-editor.org/info/rfc6760>.

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

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

[RFC8305] Schinazi, D. and T. Pauly, "Happy Eyeballs Version 2: Better Connectivity Using Concurrency", RFC 8305, DOI 10.17487/RFC8305, December 2017, <https://www.rfc-editor.org/info/rfc8305>.

[RFC8375] Pfister, P. and T. Lemon, "Special-Use Domain 'home.arpa.'", RFC 8375, DOI 10.17487/RFC8375, May 2018, <https://www.rfc-editor.org/info/rfc8375>.

[Broker] Cheshire, S. and T. Lemon, "Service Discovery Broker", drdraft-sctl-discovery-broker-00 (work in progress), July 2017.

[DisProx] Cheshire, S., "Discovery Proxy for Multicast DNS-Based Service Discovery", draft-ietf-dnssd-hybrid-08 (work in progress), March 2018.

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

[DNS-UL] Sekar, K., "Dynamic DNS Update Leases", draft-sekar-dns-ul-01 (work in progress), August 2006.

[DSO] Bellis, R., Cheshire, S., Dickinson, J., Dickinson, S., Lemon, T., and T. Pusateri, "DNS Stateful Operations", draft-ietf-dnsop-session-signal-07 (work in progress), March 2018.

[Owner] Cheshire, S. and M. Krochmal, "EDNS0 OWNER Option", draft-cheshire-edns0-owner-option-01 (work in progress), July 2017.

[Push] Pusateri, T. and S. Cheshire, "DNS Push Notifications", draft-ietf-dnssd-push-14 (work in progress), March 2018.

[RegProt] Cheshire, S. and T. Lemon, "Service Registration Protocol for DNS-Based Service Discovery", draft-sctl-service-registration-00 (work in progress), July 2017.

[Relay] Cheshire, S. and T. Lemon, "Multicast DNS Discovery Relay", draft-sctl-dnssd-mdns-relay-04 (work in progress), March 2018.

[STR] "Service Name and Transport Protocol Port Number Registry", <http://www.iana.org/assignments/service-names-port-numbers/>.

[SUDN] "Special-Use Domain Names Registry", <https://www.iana.org/assignments/special-use-domain-names/>.

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

Appendix A. IETF Terminal Room Printer Discovery Walk-Through

For about a decade now, the talented 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 either way the process by which clients query to discover services is the same.

This appendix gives a detailed step-by step account of how this client query process works. It starts with a client joining the Wi-Fi network and doing a DHCP request, and ends with paper coming out of the printer. The reason the explanation is gives the specific details of every step is to avoid inadvertently having a hand-waving "and then a miracle occurs" part, which misses out some important detail. 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 0A | xxd -r -p
meeting.ietf.org
```

A.1. Domain Enumeration using PTR queries

Our DHCP domain name is meeting.ietf.org. Does meeting.ietf.org recommend that we look in any Wide Area Service Discovery domains? This step is called Domain Enumeration [RFC6763], and is performed using a DNS PTR query for a name with the special prefix "lb._dns-sd._udp":

```
% 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 in the Answer Section you'll see that the answer to the PTR query 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 PTR query to Google's 8.8.8.8 public resolver:

```
% 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)
;; WHEN: Wed Mar 13 10:18:27 2013
;; MSG SIZE  rcvd: 64
```

In the Answer Section 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.tampa1.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)  48.852 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.2. Instance Enumeration using PTR queries on a macOS computer

Once one or more service discovery domains have been determined, the client then looks for instances of the desired service type. This step is called Instance Enumeration and is also performed using a DNS PTR queries, using a name with a prefix indicating the type of service that is being sought.

A macOS computer with appropriate printer drivers installed will look for instances of the service type "_pdl-datastream._tcp" in the domain "meeting.ietf.org", as shown below. This is typically performed just once, the first time the macOS computer is set up to use that printer.

```
% 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.

A.3. Printing from a macOS computer

When the user actually prints something, macOS sends a DNS SRV query for the printer name learned in the previous Instance Enumeration step, to learn the target host and port for the service. This DNS SRV query is then followed by address queries for the target host's IPv4 and/or IPv6 addresses. The necessary address records are usually included in the Additional Section of the reply to the SRV query, so that these address queries can be answered from the local cache, without resulting in additional packets over the air.

```
% 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.

A.4. Instance Enumeration using PTR queries on an iOS device

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). Consequently, the iOS device sends its Instance Enumeration DNS PTR queries using the prefix "_universal._sub._ipp._tcp" to indicate that it is looking for the subset of IPP printers that support Universal Raster Format.

```
% 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 URF-capable 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.

A.5. Printing from an iOS device

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

```
% 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 from the macOS example, 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

ANIMA WG T. Eckert
Internet-Draft Huawei
Intended status: Standards Track Jul 1, 2018
Expires: January 2, 2019

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 fetures 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 possibel 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 reuseable 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 will expire on January 2, 2019.

Copyright Notice

Table of Contents

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 fetures 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 possibel 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 choosen 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 requireed), 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 reuseable 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 wants 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 uses 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 2.3.3for more details.

See Section 5 for the detailled 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 occurances of XXXX in rfcXXXX with the RFC number assigned to this document and remove this edit note.]

```
objective-value  /= { 1*elements }
elements        //= ( @rfcXXXX: { 1*relement } )

relement  = ( relement-codepoint => relement-value )
relement-codepoint = uint
relement-value     = any
```

If an objective wants to use reuseable elements, the objective-value MUST be a CBOR map and the reuseable elements are found under the key "@rfcXXXX". Objectives that do not want reuseable elements as defined here can use any objective-value format including a CBOR map, but they can not use the "@rfcXXXX" key if they use a map. This approach was choosen 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 reuseable 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 neworks such as in IoT.

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

Reuseable 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 reuseable 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" abtraction. 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 sring 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 unicasted). 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 unicasted "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 indiciated 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 )

context-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 neessary, 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 detailled 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 announcemenets 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 retroative 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 indended 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]
Bormann, C., Carpenter, B., and B. Liu, "A Generic Autonomic Signaling Protocol (GRASP)", draft-ietf-anima-grasp-15 (work in progress), July 2017.

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

8.2. Informative References

[I-D.ietf-anima-autonomic-control-plane]
Eckert, T., Behringer, M., and S. Bjarnason, "An Autonomic Control Plane (ACP)", draft-ietf-anima-autonomic-control-plane-16 (work in progress), June 2018.

Author’s Address

Toerless Eckert
Futurewei Technologies Inc.
2330 Central Expy
Santa Clara 95050
USA

Email: tte+ietf@cs.fau.de

DNSOP Working Group
Internet-Draft
Updates: 1035, 7766 (if approved)
Intended status: Standards Track
Expires: June 9, 2019

R. Bellis
ISC
S. Cheshire
Apple Inc.
J. Dickinson
S. Dickinson
Sinodun
T. Lemon
Nibbhaya Consulting
T. Pusateri
Unaffiliated
December 06, 2018

DNS Stateful Operations
draft-ietf-dnsop-session-signal-20

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 which has different message semantics, and a new result code. 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

Copyright Notice

Table of Contents

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.

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

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

The format for DSO messages (Section 5.4) 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. Just as in traditional DNS over TCP [RFC1035] [RFC7766] the stream protocol carrying DSO messages (which are just another kind of DNS message) frames them by putting a 16-bit message length at the start, so the length of the DSO message 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: DNS Stateful Operations.

connection: 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].

session: The unqualified term "session" in the context of this document refers to a persistent network connection between two endpoints which allows for the exchange of DNS messages over a connection where either end of the connection can send messages to the other end. (The term has no relationship to the "session layer" of the OSI "seven-layer model".)

DSO Session: a session 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].

close gracefully: a normal session shutdown, where the client closes the TCP connection to the server using a graceful close, such that no data is lost (e.g., using TCP FIN, see Section 5.3).

forcibly abort: a session shutdown as a result of a fatal error, where the TCP connection is unilaterally aborted without regard for data loss (e.g., using TCP RST, see Section 5.3).

server: the software with a listening socket, awaiting incoming connection requests, in the usual DNS sense.

client: the software which initiates a connection to the server's listening socket, in the usual DNS sense.

initiator: the software which sends a DSO request message or a DSO unidirectional message during a DSO session. Either a client or server can be an initiator

responder: the software which receives a DSO request message or a DSO unidirectional message during a DSO

session. Either a client or server can be a responder.

sender: the software which is sending a DNS message, a DSO message, a DNS response, or a DSO response.

receiver: the software which is receiving a DNS message, a DSO message, a DNS response, or a DSO response.

service instance: a specific instance of server software running on a specific host (Section 9.1).

long-lived operation: a long-lived operation is an outstanding operation on a DSO session where either the client or server, acting as initiator, has requested that the responder send new information regarding the request, as it becomes available.

Early Data: A TLS 1.3 handshake containing early data that begins a DSO session ([RFC8446] section 2.3). TCP Fast Open is only permitted when using TLS.

DNS message: any DNS message, including DNS queries, response, updates, DSO messages, etc.

DNS request message: any DNS message where the QR bit is 0.

DNS response message: any DNS message where the QR bit is 1.

DSO message: a DSO request message, DSO unidirectional message, or a DSO response to a DSO request message. If the QR bit is 1 in a DSO message, it is a DSO response message. If the QR bit is 0 in a DSO message, it is a DSO request message or DSO unidirectional message, as determined by the specification of its primary TLV.

DSO response message: a response to a DSO request message.

DSO request message: a DSO message that requires a response.

DSO unidirectional message: a DSO message that does not require and cannot induce a response.

Primary TLV: The first TLV in a DSO message or DSO response; in the DSO message this determines the nature of the operation being performed.

Additional TLV: Any TLVs in a DSO message response that follow the primary TLV.

Response Primary TLV: The (optional) first TLV in a DSO response.

Response Additional TLV: Any TLVs in a DSO response that follow the (optional) Response Primary TLV.

inactivity timer: the time since the most recent non-keepalive DNS message was sent or received. (see Section 6.4)

keepalive timer: the time since the most recent DNS message was sent or received. (see Section 6.5)

session timeouts: the inactivity timer and the keepalive timer.

inactivity timeout: the maximum value that the inactivity timer can have before the connection is gracefully closed.

keepalive interval: the maximum value that the keepalive timer can have before the client is required to send a keepalive. (see Section 7.1)

resetting a timer: setting the timer value to zero and restarting the timer.

clearing a timer: setting the timer value to zero but not restarting the timer.

4. Applicability

DNS Stateful Operations are applicable to several known use cases and are only applicable on transports that are capable of supporting a DSO Session.

4.1. Use Cases

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

4.1.1. Session Management

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.

4.1.2. Long-lived Subscriptions

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.

4.2. Applicable Transports

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 DNS Stateful Operations 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 mid-connection (see [RFC7858] section 7). A connection is either DNS-over-TCP from the start, or DNS-over-TLS from the start.

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.

5. Protocol Details

The overall flow of DNS Stateful Operations goes through a series of phases:

Connection Establishment: A client establishes a connection to a server. (Section 4.2)

Connected but sessionless: A connection exists, but a DSO session has not been established. DNS messages can be sent from the client to server, and DNS responses can be sent from servers to clients. In this state a client that wishes to use DSO can attempt to establish a DSO session (Section 5.1). Standard DNS-over-TCP inactivity timeout handling is in effect [RFC7766] (see Section 7.1.2).

DSO Session Establishment in Progress: A client has sent a DSO request, but has not yet received a DSO response. In this phase, the client may send more DSO requests and more DNS requests, but MUST NOT send DSO unidirectional messages (Section 5.1).

DSO Session Establishment Failed: The attempt to establish the DSO session did not succeed. At this point, the client is permitted to continue operating without a DSO session (Connected but Sessionless) but does not send further DSO messages (Section 5.1).

DSO Session Established: Both client and server may send DSO messages and DNS messages; both may send replies in response to messages they receive (Section 5.2). The inactivity timer (Section 6.4) is active; the keepalive timer (Section 6.5) is active. Standard DNS-over-TCP inactivity timeout handling is no longer in effect [RFC7766] (see Section 7.1.2).

Server Shutdown: The server has decided to gracefully terminate the session, and has sent the client a Retry Delay message (Section 6.6.1). There may still be unprocessed messages from the client; the server will ignore these. The server will not send any further messages to the client (Section 6.6.1.1).

Client Shutdown: The client has decided to disconnect, either because it no longer needs service, the connection is inactive (Section 6.4.1), or because the server sent it a Retry Delay message (Section 6.6.1). The client closes the connection gracefully Section 5.3.

Reconnect: The client disconnected as a result of a server shutdown. The client either waits for the server-specified Retry Delay to expire (Section 6.6.3), or else contacts a different server instance. If the client no longer needs service, it does not reconnect.

Forcibly Abort: The client or server detected a protocol error, and further communication would have undefined behavior. The client or server forcibly aborts the connection (Section 5.3).

Abort Reconnect Wait: The client has forcibly aborted the connection, but still needs service. Or, the server forcibly aborted the connection, but the client still needs service. The client either connects to a different service instance (Section 9.1) or waits to reconnect (Section 6.6.3.1).

5.1. 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 4).

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 unidirectional 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 unidirectional 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 7.1), and receiving a response, with matching MESSAGE ID, and RCODE set to NOERROR (0), indicating that the DSO request was successful.

Some DSO messages are permitted as early data (Section 11.1). Others are not. Unidirectional messages are never permitted as early data unless an implicit session exists.

If a server receives a DSO message in early data whose primary TLV is not permitted to appear in early data, the server MUST forcibly abort the connection. If a client receives a DSO message in early data, and there is no implicit DSO session, the client MUST forcibly abort the connection. This can only be enforced on TLS connections; therefore, servers MUST NOT enable TFO when listening for a connection that does not require TLS.

5.1.1. Session Establishment Failure

If the response RCODE is set to NOTIMP (4), or in practise any value other than NOERROR (0) or DSOTYPENI (defined below), 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.

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.

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 that service instance 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 (Section 6.6.3.2).

In the second case, the client SHOULD wait 30 seconds, after which time the server will be assumed not to support DSO. If the server doesn't respond within 30 seconds, the client MUST forcibly abort 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 (Section 6.6.3.1).

5.1.2. Session Establishment Success

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.

5.2. Operations After Session Establishment

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 a server that supports DNS Stateful Operations MUST return an RCODE of NOERROR when it receives a Keepalive TLV DSO request message, the Keepalive TLV is an ideal candidate for use in establishing a DSO session. Any other option that can only succeed when sent to a server of the desired kind is also a good candidate for use in establishing 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.

5.3. Session Termination

A "DSO Session" is terminated when the underlying connection is closed. Sessions are "closed gracefully" as a result of the server closing a session because it is overloaded, the client closing the session because it is done, or the client closing the session because it is inactive. Sessions are "forcibly aborted" when either the client or server closes the connection because of a protocol error.

- 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.

5.3.1. Handling Protocol Errors

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 compliant DSO implementation. 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".

5.4. Message Format

A DSO message begins with the standard twelve-byte DNS message header [RFC1035] with the OPCODE field set to the DSO OPCODE. 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.

```
                                            1   1   1   1   1   1
      0   1   2   3   4   5   6   7   8   9   0   1   2   3   4   5
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    |                          MESSAGE ID                           |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    |QR |    OPCODE     |            Z              |     RCODE     |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    |                    QDCOUNT (MUST be zero)                     |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    |                    ANCOUNT (MUST be zero)                     |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    |                    NSCOUNT (MUST be zero)                     |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    |                    ARCOUNT (MUST be zero)                     |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    |                                                               |
    /                           DSO Data                            /
    /                                                               /
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

5.4.1. DNS Header Fields in DSO Messages

In a DSO unidirectional message the MESSAGE ID field MUST be set to zero. In a DSO 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 DSO request message 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 a DSO request message or a DSO unidirectional message is determined only by the specification for the Primary TLV. An acknowledgment cannot be requested by including a nonzero message ID in a message that is required according to its primary TLV to be unidirectional. Nor can an acknowledgment be prevented by sending a message ID of zero in a message that is required to be a DSO request message according to its primary TLV. A responder that receives either such malformed message MUST treat it as a fatal error and forcibly abort the connection immediately.

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

In a DSO 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 DSO response message (QR=1) the MESSAGE ID field MUST contain a copy of the value of the MESSAGE ID field in the DSO request message being responded to. In a DSO response message (QR=1) the MESSAGE ID field MUST NOT be zero. If a DSO 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.

The Z bits are currently unused in DSO messages, and in both DSO request messages and DSO responses the Z bits MUST be set to zero (0) on transmission and MUST be ignored on reception.

In a DSO request message (QR=0) the RCODE is set according to the definition of the request. For example, in a Retry Delay message (Section 6.6.1) the RCODE indicates the reason for termination. However, in most cases, except where clearly specified otherwise, in a DSO 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:

Code	Mnemonic	Description
0	NOERROR	Operation processed successfully
1	FORMERR	Format error
2	SERVFAIL	Server failed to process DSO request message due to a problem with the server
4	NOTIMP	DSO not supported
5	REFUSED	Operation declined for policy reasons
[TBA2] 11	DSOTYPENI	Primary TLV's DSO-Type is not implemented

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 response codes not defined here, then that document MUST specify the specific interpretation of those RCODE values in the context of that new DSO TLV.

5.4.2. 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 in Section 5.4.3.

A DSO request message or DSO unidirectional message MUST contain at least one TLV. The first TLV in a DSO request message or DSO unidirectional message is referred to as the "Primary TLV" and determines the nature of the operation being performed, including whether it is a DSO request or a DSO unidirectional operation. In some cases it may be appropriate to include other TLVs in a DSO request message or DSO unidirectional message, such as the Encryption Padding TLV (Section 7.3), 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 8.2 for more information.

A DSO response message may contain one or more TLVs with the Primary TLV DSO-TYPE the same as the Primary TLV from the corresponding DSO request message or it may contain zero or more Additional TLVs only. 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 DSO request messages, which generate corresponding DSO responses. In some specialized high-traffic use cases, it may be appropriate to specify DSO unidirectional messages. DSO unidirectional messages can be more efficient on the network, because they don't generate a stream of corresponding reply messages. Using DSO unidirectional 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 DSO request message (i.e., QR=0), that message is to be unidirectional, the MESSAGE ID field MUST be set to zero and the receiver MUST NOT generate any response message corresponding to this DSO unidirectional message.

The previous point, that the receiver MUST NOT generate responses to DSO unidirectional 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.

DSO unidirectional 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. DSO unidirectional 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 DSO unidirectional 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 DSO unidirectional 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].

5.4.3. 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 DSO message (i.e., QR=0), that DSO message is unidirectional or is a request message which requires a response. If the DSO message requires a response, 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.

```
                                           1   1   1   1   1   1
       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.

5.4.3.1. Request TLVs

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

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

5.4.3.2. Response TLVs

Depending on the operation, a DSO response message MAY contain no TLVs, because it is simply a response to a previous DSO request message, and the MESSAGE ID in the header is sufficient to identify the DSO 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 DSO request message 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.

5.4.3.3. 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. The error response MUST NOT contain a copy of the unrecognized Primary TLV.

If DSO unidirectional 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 DSO unidirectional 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.

5.4.4. EDNS(0) and TSIG

Since the ARCOUNT field MUST be zero, a DSO message cannot 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 7.3 MUST be used.

A DSO message can't contain a TSIG record, because a TSIG record is included in the additional section of the message, which would mean that ARCOUNT would be greater than zero. DSO messages are required to have an ARCOUNT of zero. Therefore, if use of signatures with DSO messages becomes necessary in the future, a new DSO TLV would have 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.

5.5. Message Handling

As described above in Section 5.4.1, whether an outgoing DSO message with the QR bit in the DNS header set to zero is a DSO request or DSO unidirectional message 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.

Every DSO message with the QR bit in the DNS header set to zero and a nonzero MESSAGE ID field is a DSO request message, and MUST elicit a corresponding response, with the QR bit in the DNS header set to one and the MESSAGE ID field set to the value given in the corresponding DSO request message.

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.

Every DSO message with both the QR bit in the DNS header and the MESSAGE ID field set to zero is a DSO unidirectional message, and MUST NOT elicit a response.

5.5.1. Delayed Acknowledgement Management

Generally, most good TCP implementations employ a delayed acknowledgement timer to provide more efficient use of the network and better performance.

With a bidirectional exchange over TCP, as for example with a DSO request message, the operating system TCP implementation waits for the application-layer client software to generate the corresponding DSO response message. It can then send a single combined packet containing the TCP acknowledgement, the TCP window update, and the application-generated DSO response message. This is more efficient than sending three separate packets, as would occur if the TCP packet containing the DSO request were acknowledged immediately.

With a DSO unidirectional message or DSO response message, there is no corresponding application-generated DSO response message, and consequently, no hint to the transport protocol about when it should send its acknowledgement and window update.

Some networking APIs provide a mechanism that allows the application-layer client software to signal to the transport protocol that no response will be forthcoming (in effect it can be thought of as a zero-length "empty" write). Where available in the networking API being used, the recipient of a DSO unidirectional message or DSO response message, having parsed and interpreted the message, SHOULD then use this mechanism provided by the networking API to signal that no response for this message will be forthcoming, so that the TCP implementation can go ahead and send its acknowledgement and window update without further delay. See Section 9.5 for further discussion of why this is important.

5.5.2. MESSAGE ID Namespaces

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 DSO 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 0x00001-0x0FFFF 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 5.4.1, an initiator MUST NOT reuse a MESSAGE ID that it already has in use for an outstanding DSO request message (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 DSO request message 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 DSO request message (QR=0) where the MESSAGE ID is not zero, and the responder tracks request MESSAGE IDs, and the MESSAGE ID matches the MESSAGE ID of a DSO request message 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 responder 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.

5.5.3. Error Responses

When a DSO unidirectional message type is received (MESSAGE ID field is zero), the receiver should already be expecting this DSO message type. Section 5.4.3.3 describes the handling of unknown DSO message types. Parsing errors MUST also result in the receiver forcibly aborting the connection. When a DSO unidirectional message of an unexpected type is received, the receiver SHOULD forcibly abort the connection. Whether the connection should be forcibly aborted for other internal errors processing the DSO unidirectional message is implementation dependent, according to the severity of the error.

When a 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 6.6.1.

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 6.6.1) then the client SHOULD continue to use that DSO Session for subsequent operations.

5.6. 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].

Long-lived operations, if successful, will remain 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 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.

6. DSO Session Lifecycle and Timers

6.1. DSO Session Initiation

A DSO Session begins as described in Section 5.1.

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.

The server MUST act on messages in the order they are received, 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.

Section 6.2.1.1 of the DNS-over-TCP specification [RFC7766] specifies this in more detail.

6.2. 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 7.1).

The first timeout value, the inactivity timeout, is the maximum time for which a client may speculatively keep an inactive DSO Session open 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 active 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 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.

6.3. Inactive DSO Sessions

At both servers and clients, the generation or reception of any complete DNS message (including DNS requests, responses, updates, DSO messages, etc.) resets both timers for that DSO Session, with the one 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, i.e. 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.

6.4. The Inactivity Timeout

The purpose of the inactivity timeout is for the server to balance the 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.

6.4.1. 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 7.1.

6.4.2. 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 forcibly 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 0xFFFFFFFE (2^32-2 milliseconds, approximately 49.7 days).

6.5. The Keepalive Interval

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

6.5.1. 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 7.1). 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 uncancelled, 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.

6.5.2. 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. Moreover for a keep-alive value that is smaller than the time needed for the transport to retransmit, a single packet loss would cause a server to overzealously abort the connect. For example, a (hypothetical and unrealistic) keepalive interval value of 100 ms would result in a continuous stream of ten messages per second or more (if allowed by the current congestion control window), in both directions, to keep the DSO Session alive. And, in this extreme example, a single retransmission over a path with, e.g., 100ms RTT would introduce a momentary pause in the stream of messages, long enough to cause the server to 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 0xFFFFFFFE (2^32-2 milliseconds, approximately 49.7 days).

6.6. Server-Initiated Session Termination

In addition to cancelling individual long-lived operations selectively (Section 5.6) 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:

- o The server application software or underlying operating system is shutting down or restarting.

- o The server application software terminates unexpectedly (perhaps due to a bug that makes it crash, causing the underlying operating system to send a TCP RST).

- o 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.

- o The client fails to meets 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 6.2).

- o The client sends a grossly invalid or malformed request that is indicative of a seriously defective client implementation.

- o The server is over capacity and needs to shed some load.

6.6.1. 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 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 7.2. 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.

6.6.1.1. 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).

6.6.2. Misbehaving Clients

A server may determine that a client is not following the protocol correctly. There may be no way for the server to recover the session, in which case the server forcibly terminates the connection. Since the client doesn't know why the connection dropped, it may reconnect immediately. If the server has determined that a client is not following the protocol correctly, it may terminate the DSO session as soon as it is established, specifying a long retry-delay to prevent the client from immediately reconnecting.

6.6.3. 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.

6.6.3.1. Reconnecting After a Forcible Abort

If a connection was forcibly aborted by the client, the client SHOULD mark that service instance as not supporting DSO. The client MAY reconnect but not attempt to use DSO, or may connect to a different service instance, if applicable.

6.6.3.2. Reconnecting After an Unexplained Connection Drop

It is also possible for a server to forcibly terminate the connection; in this case the client doesn’t know whether the termination was the result of a protocol error or a network outage. When the client notices that the connection has been dropped, it can attempt to reconnect immediately. However, if the connection is dropped again without the client being able to successfully do whatever it is trying to do, it should mark the server as not supporting DSO.

6.6.3.3. Probing for Working DSO Support

Once a server has been marked by the client as not supporting DSO, the client SHOULD NOT attempt DSO operations on that server until some time has elapsed. A reasonable minimum would be an hour. Since forcibly aborted connections are the result of a software failure, it’s not likely that the problem will be solved in the first hour after it’s first encountered. However, by restricting the retry interval to an hour, the client will be able to notice when the problem has been fixed without placing an undue burden on the server.

7. Base TLVs for DNS Stateful Operations

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

7.1. Keepalive TLV

The Keepalive TLV (DSO-TYPE=1) performs two functions. Primarily it establishes the values for the Session Timeouts. Incidentally, it also resets the keepalive timer for the DSO Session, meaning that it can be used as a kind of "no-op" message for the purpose of keeping a session alive. 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.

DSO messages with the Keepalive TLV as the primary TLV may appear in early data.

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

```
                     1 1 1 1 1 1 1 1 1 1 2 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 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 itself 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 DSO request message, 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 Timeouts 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 request messages 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 5.1) a DSO Keepalive message MAY be initiated by a server. When sent by a server, the DSO Keepalive message MUST be sent as a DSO unidirectional 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 DSO Keepalive unidirectional 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 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 DSO Keepalive unidirectional 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 DSO Keepalive request message is the only way for a client to initiate a DSO Session.

7.1.1. 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. The act of receiving the message containing the DSO Keepalive TLV itself resets the keepalive timer, and 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.

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.

- o 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.

- o 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.

- o 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.

- o 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.

7.1.2. Relationship 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.

7.2. Retry Delay TLV

The Retry Delay TLV (DSO-TYPE=2) can be used as a Primary TLV (unidirectional) in a server-to-client message, or as a Response Additional TLV in either direction. DSO messages with a Relay Delay TLV as their primary TLV are not permitted in early data.

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

```
                     1 1 1 1 1 1 1 1 1 1 2 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 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 6.6.1.

7.2.1. 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 a DSO unidirectional 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 6.6.1. The RCODE in the message header SHOULD indicate the principal reason for the termination:

- o NOERROR indicates a routine shutdown or restart.
- o FORMERR indicates that a client request was too badly malformed for the session to continue.
- o SERVFAIL indicates that the server is overloaded due to resource exhaustion and needs to shed load.
- o 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.
- o 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 is 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 the 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 a DSO unidirectional 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.

7.2.2. Retry Delay TLV used as a Response Additional TLV

In the case of a DSO request message that results in 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.

7.3. 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 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).

```
                                           1   1   1   1   1   1
     0   1   2   3   4   5   6   7   8   9   0   1   2   3   4   5
   +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
   /                                                               /
   /              PADDING -- 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 message, response, or both. As specified for the EDNS(0) Padding Option [RFC7830] if a DSO request message is received with an Encryption Padding TLV, then the DSO 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].

8. Summary Highlights

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

8.1. 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 Unidirectional 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:

	MESSAGE ID zero	MESSAGE ID nonzero
QR=0	DSO unidirectional Message	DSO Request Message
QR=1	Invalid - Fatal Error	DSO Response Message

8.2. 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 DSO requests or DSO unidirectional 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 Unidirectional 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 a DSO request message or DSO unidirectional 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: DSO requests or DSO unidirectional 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 Unidirectional 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 a DSO request message or DSO unidirectional 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.

	C-P	C-U	C-A	CRP	CRA	S-P	S-U	S-A	SRP	SRA
KeepAlive	X			X			X			
RetryDelay					X		X			X
Padding			X		X			X		X

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.

9. Additional Considerations

9.1. Service Instances

We use the term service instance to refer to software running on a host which can receive connections on some set of IP address and port tuples. What makes the software an instance is that regardless of which of these tuples the client uses to connect to it, the client is connected to the same software, running on the same node (but see Section 9.2), and will receive the same answers and the same keying information.

Service instances are identified from the perspective of the client. If the client is configured with IP addresses and port number tuples, it has no way to tell if the service offered at one tuple is the same server that is listening on a different tuple. So in this case, the client treats each such tuple as if it references a separate service instance.

In some cases a client is configured with a hostname and a port number (either implicitly, where the port number is omitted and assumed, or explicitly, as in the case of DNS SRV records). In these cases, the (hostname, port) tuple uniquely identifies the service instance (hostname comparisons are case-insensitive [RFC1034].

It is possible that two hostnames might point to some common IP addresses; this is a configuration error which the client is not obliged to detect. The effect of this could be that after being told to disconnect, the client might reconnect to the same server because it is represented as a different service instance.

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".

9.2. Anycast Considerations

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.

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.

9.3. 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 service instance (see Section 9.1), the client SHOULD use a single shared DSO Session for all those operations.

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.

9.4. Operational Considerations for Middlebox

Where an application-layer middlebox (e.g., a DNS proxy, forwarder, or session multiplexer) is in the path, care must be taken to avoid a configuration in which DSO traffic is mis-handled. The simplest way to avoid such problems is to avoid using middleboxes. When this is not possible, middleboxes should be evaluated to make sure that they behave correctly.

Correct behavior for middleboxes consists of one of:

- The middlebox does not forward DSO messages, and responds to DSO messages with a response code other than NOERROR or DSOTYPENI.

- The middlebox acts as a DSO server and follows this specification in establishing connections.

- There is a 1:1 correspondence between incoming and outgoing connections, such that when a connection is established to the middlebox, it is guaranteed that exactly one corresponding connection will be established from the middlebox to some DNS resolver, and all incoming messages will be forwarded without modification or reordering. An example of this would be a NAT forwarder or TCP connection optimizer (e.g. for a high-latency connection such as a geosynchronous satellite link).

Middleboxes that do not meet one of the above criteria are very likely to fail in unexpected and difficult-to-diagnose ways. For example, a DNS load balancer might unbundle DNS messages from the incoming TCP stream and forward each message from the stream to a different DNS server. If such a load balancer is in use, and the DNS servers it points implement DSO and are configured to enable DSO, DSO session establishment will succeed, but no coherent session will exist between the client and the server. If such a load balancer is pointed at a DNS server that does not implement DSO or is configured not to allow DSO, no such problem will exist, but such a configuration risks unexpected failure if new server software is installed which does implement DSO.

It is of course possible to implement a middlebox that properly supports DSO. It is even possible to implement one that implements DSO with long-lived operations. This can be done either by maintaining a 1:1 correspondence between incoming and outgoing connections, as mentioned above, or by terminating incoming sessions at the middlebox, but maintaining state in the middlebox about any long-lived that are requested. Specifying this in detail is beyond the scope of this document.

9.5. TCP Delayed Acknowledgement Considerations

Most modern implementations of the Transmission Control Protocol (TCP) include a feature called "Delayed Acknowledgement" [RFC1122].

Without this feature, TCP can be very wasteful on the network. For illustration, consider a simple example like remote login, using a very simple TCP implementation that lacks delayed acks. When the user types a keystroke, a data packet is sent. When the data packet arrives at the server, the simple TCP implementation sends an immediate acknowledgement. Mere milliseconds later, the server process reads the one byte of keystroke data, and consequently the simple TCP implementation sends an immediate window update. Mere milliseconds later, the server process generates the character echo, and sends this data back in reply. The simple TCP implementation then sends this data packet immediately too. In this case, this simple TCP implementation sends a burst of three packets almost instantaneously (ack, window update, data).

Clearly it would be more efficient if the TCP implementation were to combine the three separate packets into one, and this is what the delayed ack feature enables.

With delayed ack, the TCP implementation waits after receiving a data packet, typically for 200 ms, and then send its ack if (a) more data packet(s) arrive (b) the receiving process generates some reply data, or (c) 200 ms elapses without either of the above occurring.

With delayed ack, remote login becomes much more efficient, generating just one packet instead of three for each character echo.

The logic of delayed ack is that the 200 ms delay cannot do any significant harm. If something at the other end were waiting for something, then the receiving process should generate the reply that the thing at the end is waiting for, and TCP will then immediately send that reply (and the ack and window update). And if the receiving process does not in fact generate any reply for this particular message, then by definition the thing at the other end cannot be waiting for anything, so the 200 ms delay is harmless.

This assumption may be true, unless the sender is using Nagle's algorithm, a similar efficiency feature, created to protect the network from poorly written client software that performs many rapid small writes in succession. Nagle's algorithm allows these small writes to be combined into larger, less wasteful packets.

Unfortunately, Nagle's algorithm and delayed ack, two valuable efficiency features, can interact badly with each other when used together [NagleDA].

DSO request messages elicit responses; DSO unidirectional messages and DSO response messages do not.

For DSO request messages, which do elicit responses, Nagle's algorithm and delayed ack work as intended.

For DSO messages that do not elicit responses, the delayed ack mechanism causes the ack to be delayed by 200 ms. The 200 ms delay on the ack can in turn cause Nagle's algorithm to prevent the sender from sending any more data for 200 ms until the awaited ack arrives. On an enterprise GigE backbone with sub-millisecond round-trip times, a 200 ms delay is enormous in comparison.

When this issues is raised, there are two solutions that are often offered, neither of them ideal:

1. Disable delayed ack. For DSO messages that elicit no response, removing delayed ack avoids the needless 200 ms delay, and sends back an immediate ack, which tells Nagle's algorithm that it should immediately grant the sender permission to send its next packet. Unfortunately, for DSO messages that *do* elicit a response, removing delayed ack removes the efficiency gains of combining acks with data, and the responder will now send two or three packets instead of one.

2. Disable Nagle's algorithm. When acks are delayed by the delayed ack algorithm, removing Nagle's algorithm prevents the sender from being blocked from sending its next small packet immediately. Unfortunately, on a network with a higher round-trip time, removing Nagle's algorithm removes the efficiency gains of combining multiple small packets into fewer larger ones, with the goal of limiting the number of small packets in flight at any one time.

For DSO messages that elicit a response, delayed ack and Nagle's algorithm do the right thing.

The problem here is that with DSO messages that elicit no response, the TCP implementation is stuck waiting, unsure if a response is about to be generated, or whether the TCP implementation should go ahead and send an ack and window update.

The solution is networking APIs that allow the receiver to inform the TCP implementation that a received message has been read, processed, and no response for this message will be generated. TCP can then stop waiting for a response that will never come, and immediately go ahead and send an ack and window update.

For implementations of DSO, disabling delayed ack is NOT RECOMMENDED, because of the harm this can do to the network.

For implementations of DSO, disabling Nagle's algorithm is NOT RECOMMENDED, because of the harm this can do to the network.

At the time that this document is being prepared for publication, it is known that at least one TCP implementation provides the ability for the recipient of a TCP message to signal that it is not going to send a response, and hence the delayed ack mechanism can stop waiting. Implementations on operating systems where this feature is available SHOULD make use of it.

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:

Type	Name	Early Data	Status	Reference
0000	Reserved	NO	Standard	RFC-TBD
0001	KeepAlive	OK	Standard	RFC-TBD
0002	RetryDelay	NO	Standard	RFC-TBD
0003	EncryptionPadding	NA	Standard	RFC-TBD
0004-003F	Unassigned, reserved for DSO session-management TLVs	NO		
0040-F7FF	Unassigned	NO		
F800-FBFF	Experimental/local use	NO		
FC00-FFFF	Reserved for future expansion	NO		

The meanings of the fields are as follows:

Type: the 16-bit DSO type code

Name: the human-readable name of the TLV

Early Data: If OK, this TLV may be sent as early data in a TLS 0-RTT ([RFC8446] Section 2.3) initial handshake. If NA, the TLV may appear as a secondary TLV in a DSO message that is send as early data.

Status: IETF Document status (or "External" if not documented in an IETF document.

Reference: A stable reference to the document in which this TLV is defined.

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" range FC00-FFFF require publication of an IETF Standards Action document [RFC8126].

Any document defining a new TLV which lists a value of "OK" in the 0-RTT column must include a threat analysis for the use of the TLV in the case of TLS 0-RTT. See Section 11.1 for details.

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 unidirectional DSO Retry Delay 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 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. TLS 0-RTT Considerations

DSO permits zero round-trip operation using TCP Fast Open [RFC7413] with TLS 1.3 [RFC8446] 0-RTT to reduce or eliminate round trips in session establishment. TCP Fast Open is only permitted in combination with TLS 0-RTT. In the rest of this section we refer to TLS 1.3 early data in a TLS 0-RTT initial handshake message, whether or not it is included in a TCP SYN packet with early data using the TCP Fast Open option, as "early data."

A DSO message may or may not be permitted to be sent as early data. The definition for each TLV that can be used as a primary TLV is required to state whether or not that TLV is permitted as early data. Only response-requiring messages are ever permitted as early data, and only clients are permitted to send any DSO message as early data, unless there is an implicit session (see Section 5.1).

For DSO messages that are permitted as early data, a client MAY include one or more such messages as early data without having to wait for a DSO response to the first DSO request message to confirm successful establishment of a DSO session.

However, unless there is an implicit session, a client MUST NOT send DSO unidirectional messages until after a DSO Session has been mutually established.

Similarly, unless there is an implicit session, 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 as early data 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.

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, that TLV must be specified as not permitted in 0-RTT messages. This prevents work from being done until a round-trip has occurred from the server to the client to verify that the source address of the packet is reachable.

Documents that define new TLVs must state whether each new TLV may be sent as early data. Such documents must include a threat analysis in the security considerations section for each TLV defined in the document that may be sent as early data. This threat analysis should be done based on the advice given in [RFC8446] Section 2.3, 8 and Appendix E.5.

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, Bernie Volz and Bob Harold for their helpful contributions to this document.

13. References

13.1. Normative References

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

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

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

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

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

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

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

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

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

13.2. Informative References

[I-D.ietf-dnsop-no-response-issue]
Andrews, M. and R. Bellis, "A Common Operational Problem in DNS Servers - Failure To Respond.", draft-ietf-dnsop-no-response-issue-12 (work in progress), November 2018.

[I-D.ietf-dnssd-mdns-relay]
Lemon, T. and S. Cheshire, "Multicast DNS Discovery Relay", draft-ietf-dnssd-mdns-relay-01 (work in progress), July 2018.

[I-D.ietf-dnssd-push]
Pusateri, T. and S. Cheshire, "DNS Push Notifications", draft-ietf-dnssd-push-16 (work in progress), November 2018.

[I-D.ietf-doh-dns-over-https]
Hoffman, P. and P. McManus, "DNS Queries over HTTPS (DoH)", draft-ietf-doh-dns-over-https-14 (work in progress), August 2018.

[I-D.ietf-dprive-padding-policy]
Mayrhofer, A., "Padding Policy for EDNS(0)", draft-ietf-dprive-padding-policy-06 (work in progress), July 2018.

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

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <https://www.rfc-editor.org/info/rfc1122>.

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

[RFC5382] Guha, S., Ed., Biswas, K., Ford, B., Sivakumar, S., and P. Srisuresh, "NAT Behavioral Requirements for TCP", BCP 142, RFC 5382, DOI 10.17487/RFC5382, October 2008, <https://www.rfc-editor.org/info/rfc5382>.

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

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

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <https://www.rfc-editor.org/info/rfc7413>.

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

[RFC7857] Penno, R., Perreault, S., Boucadair, M., Ed., Sivakumar, S., and K. Naito, "Updates to Network Address Translation (NAT) Behavioral Requirements", BCP 127, RFC 7857, DOI 10.17487/RFC7857, April 2016, <https://www.rfc-editor.org/info/rfc7857>.

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

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

Authors' Addresses

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

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

Stuart Cheshire
Apple Inc.
One Apple Park Way
Cupertino CA 95014
USA

Phone: +1 (408) 996-1010
Email: cheshire@apple.com

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

Email: jad@sinodun.com

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

Email: sara@sinodun.com

Ted Lemon
Nibbhaya Consulting
P.O. Box 958
Brattleboro VT 05302-0958
USA

Email: mellon@fugue.com

Tom Pusateri
Unaffiliated
Raleigh NC 27608
USA

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

Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: September 25, 2019

S. Cheshire
Apple Inc.

March 24, 2019

Discovery Proxy for Multicast DNS-Based Service Discovery
draft-ietf-dnssd-hybrid-10

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

Copyright Notice

Table of Contents

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 [Mcast]. 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" [RFC8375].

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 programmatically 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 |  Communcation  | Discovery  |  | Network  |  | Network  |
  | Client |---- . . . -----|   Proxy    |  | Printer  |  | Camera   |
  +--------+                +------------+  +----------+  +----------+
                                   |             |             |
                            --------------------------------------------
                           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, except 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", or a remote user sending DNS requests that cause a Discovery Proxy to perform discovery operations on its behalf.

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:

- o 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

- o 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 ends in "in-addr.arpa."

One or more DNS subdomains for IPv6 Reverse Mapping records.
   These subdomains will have names that ends 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.

```
+---------------+--------------+-------------+-------------------+
| *example.com* |  Building 1  |  1st Floor  | Alice's printer   |
|               |  Building 2  | *2nd Floor* | Bob's printer     |
|               | *Building 3* |  3rd Floor  | Charlie's printer |
|               |  Building 4  |  4th Floor  |                   |
|               |  Building 5  |             |                   |
|               |  Building 6  |             |                   |
+---------------+--------------+-------------+-------------------+
```

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, as appropriate, according to the usual protocol rules of Multicast DNS [RFC6762], for the corresponding Multicast DNS name, type and class, with the delegated zone part of the name replaced with ".local" (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, with the ".local" top-level label replaced with with the name of the delegated zone. How long the Discovery Proxy should wait to accumulate Multicast DNS responses before sending its unicast reply 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.

Note that the contents of the delegated zone, generated as it is by performing ".local" Multicast DNS queries, mirrors the records available on the local link via Multicast DNS very closely, but not precisely. There is not a full bidirectional equivalence between the two. Certain records that are available via Multicast DNS may not have equivalents in the delegated zone, possibly because they are invalid or not relevant in the delegated zone, or because they are being supressed because they are unusable outside the local link (see Section 5.5.2). Conversely, certain records that appear in the delegated zone may not have corresponding records available on the local link via Multicast DNS. In particular there are certain administrative SRV records (see Section 6) that logically fall within the delegated zone, but semantically represent metadata *about* the zone rather than records *within* the zone, and consequently these administrative records in the delegated zone do not have any corresponding counterparts in the Multicast DNS namespace of the local link.

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:

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

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

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

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

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

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

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

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

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'.

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. PTR prnt.local.
```

into this Unicast DNS response:

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

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

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 in to the Multicast DNS protocol which protect against stale data persisting for too long. When a service shuts down gracefully, it sends goodbye packets to remove its PTR records immediately from 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 [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 [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 offer a configurable option, enabled by default, to suppress Unicast DNS answers for records that are not useful outside the local link. When the option to suppress unusable records is enabled:

- o DNS A and AAAA records for IPv4 link-local addresses [RFC3927] and IPv6 link-local addresses [RFC4862] SHOULD be suppressed.

- o 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.

- o 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.

- o 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) [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.

o 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, cancel the mDNS query and 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.)

o 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.)

o 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.

o 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.

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.

Note that certain aspects of the behavior described here do not have to be implemented overtly by the Discovery Proxy; they occur naturally as a result of using existing Multicast DNS APIs.

For example, in the first case above (no LLQ or Push Notifications, and no answers in the cache) if a new Multicast DNS query is requested (either by a local client, or by the Discovery Proxy on behalf of a remote client), and there is not already an identical Multicast DNS query active, and there are no matching answers already in the Multicast DNS cache on the Discovery Proxy device, then this will cause a series of Multicast DNS query packets to be issued with exponential backoff. The exponential backoff sequence in some implementations starts at one second and then doubles for each retransmission (0, 1, 3, 7 seconds, etc.) and in others starts at one second and then triples for each retransmission (0, 1, 4, 13 seconds, etc.). In either case, if no response has been received after six seconds, that is long enough that the underlying Multicast DNS implementation will have sent three query packets without receiving any response. At that point the Discovery Proxy cancels its Multicast DNS query (so no further Multicast DNS query packets will be sent for this query) and returns a negative response to the remote client via unicast.

The six-second delay is chosen to be long enough to give enough time for devices to respond, yet short enough not to be too onerous for a human user waiting for a response. For example, using the "dig" DNS debugging tool, the current default settings result in it waiting a total of 15 seconds for a reply (three transmissions of the query packet, with a wait of 5 seconds after each packet) which is ample time for it to have received a negative reply from a Discovery Proxy after six seconds.

The statement that for a one-shot query (i.e., no LLQ or Push Notifications requested), if at least one answer is already available in the cache then a Discovery Proxy should not issue additional mDNS query packets, also occurs naturally as a result of using existing Multicast DNS APIs. If a new Multicast DNS query is requested (either locally, or by the Discovery Proxy on behalf of a remote client), for which there are relevant answers already in the Multicast DNS cache on the Discovery Proxy device, and after the answers are delivered the Multicast DNS query is then cancelled immediately, then no Multicast DNS query packets will be generated for this query.

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 NS records giving the names of all the Discovery Proxy devices on the link.

Each Discovery Proxy device MUST be configured to answer NS queries for the zone apex name by giving its own NS record, and the NS records of its fellow Discovery Proxy devices on the same link, so that it can return the correct answers for NS queries.

The target host name in the RDATA of an NS record MUST NOT reference a name that falls within any zone delegated to a Discovery Proxy. Apart from the zone apex name, all other host names that fall within a zone delegated to a Discovery Proxy correspond to local Multicast DNS host names, which logically belong to the respective Multicast DNS hosts defending those names, not the Discovery Proxy. Generally speaking, the Discovery Proxy does not own or control the delegated zone; it is merely a conduit to the corresponding ".local" namespace, which is controlled by the Multicast DNS hosts on that link. If an NS record were to reference a manually-determined host name that falls within a delegated zone, that manually-determined host name may inadvertently conflict with a corresponding ".local" host name that is owned and controlled by some device on that link.

6.3. DNS Delegation Records

Since the Multicast DNS specification [RFC6762] states that there can be no delegation (subdomains) within a ".local" namespace, this implies that any name within a zone delegated to a Discovery Proxy (except for the zone apex name itself) cannot have any answers for any DNS queries for RRTYPEs SOA, NS, or DS. Consequently:

- o for any query for the zone apex name of a zone delegated to a Discovery Proxy, the Discovery Proxy MUST generate the appropriate immediate answers as described above, and

- o for any query for RRTYPEs SOA, NS, or DS, for any name within a zone delegated to a Discovery Proxy, other than the zone apex name, instead of translating the query to its corresponding Multicast DNS ".local" equivalent, a Discovery Proxy MUST generate an immediate negative answer.

6.4. DNS SRV Records

There are certain special DNS records that logically fall within the delegated unicast DNS subdomain, but rather than mapping to their corresponding ".local" namesakes, they actually contain metadata pertaining to the operation of the delegated unicast DNS subdomain itself. They do not exist in the corresponding ".local" namespace of the local link. For these queries a Discovery Proxy MUST generate immediate answers, whether positive or negative, to avoid delays while clients wait for their query to be answered. For example, if a Discovery Proxy does not implement Long-Lived Queries [LLQ] then it MUST return an immediate negative answer to tell the client this without delay, instead of passing the query through to the local network as a query for "_dns-llq._udp.local.", and then waiting unsuccessfully for answers that will not be forthcoming.

If a Discovery Proxy implements Long-Lived Queries [LLQ] then it MUST positively respond to "_dns-llq._udp.<zone> SRV" queries, "_dns-llq._tcp.<zone> SRV" queries, and "_dns-llq-tls._tcp.<zone> SRV" queries as appropriate, else it MUST return an immediate negative answer for those queries.

If a Discovery Proxy implements DNS Push Notifications [Push] then it MUST positively respond to "_dns-push-tls._tcp.<zone>" queries, else it MUST return an immediate negative answer for those queries.

A Discovery Proxy MUST return an immediate negative answer for "_dns-update._udp.<zone> SRV" queries, "_dns-update._tcp.<zone> SRV" queries, and "_dns-update-tls._tcp.<zone> SRV" queries, since using DNS Update [RFC2136] to change zones generated dynamically from local Multicast DNS data is not possible.

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

The 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

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

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

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

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

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

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

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

[RFC4034] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Resource Records for the DNS Security Extensions", RFC 4034, DOI 10.17487/RFC4034, March 2005, <https://www.rfc-editor.org/info/rfc4034>.

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

[RFC5155] Laurie, B., Sisson, G., Arends, R., and D. Blacka, "DNS Security (DNSSEC) Hashed Authenticated Denial of Existence", RFC 5155, DOI 10.17487/RFC5155, March 2008, <https://www.rfc-editor.org/info/rfc5155>.

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

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

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

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8490] Bellis, R., Cheshire, S., Dickinson, J., Dickinson, S., Lemon, T., and T. Pusateri, "DNS Stateful Operations", RFC 8490, DOI 10.17487/RFC8490, March 2019, <https://www.rfc-editor.org/info/rfc8490>.

[Push] Pusateri, T. and S. Cheshire, "DNS Push Notifications", draft-ietf-dnssd-push-19 (work in progress), March 2019.

12.2. Informative References

[Roadmap] Cheshire, S., "Service Discovery Road Map", draft-cheshire-dnssd-roadmap-03 (work in progress), October 2018.

[DNS-UL] Sekar, K., "Dynamic DNS Update Leases", draft-sekar-dns-ul-01 (work in progress), August 2006.

[LLQ] Cheshire, S. and M. Krochmal, "DNS Long-Lived Queries", draft-sekar-dns-llq-03 (work in progress), March 2019.

[RegProt] Cheshire, S. and T. Lemon, "Service Registration Protocol for DNS-Based Service Discovery", draft-sctl-service-registration-00 (work in progress), July 2017.

[Relay] Cheshire, S. and T. Lemon, "Multicast DNS Discovery Relay", draft-sctl-dnssd-mdns-relay-04 (work in progress), March 2018.

[Mcast] Perkins, C., McBride, M., Stanley, D., Kumari, W., and J. Zuniga, "Multicast Considerations over IEEE 802 Wireless Media", draft-ietf-mboned-ieee802-mcast-problems-04 (work in progress), November 2018.

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

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

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

[RFC3492] Costello, A., "Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA)", RFC 3492, DOI 10.17487/RFC3492, March 2003, <https://www.rfc-editor.org/info/rfc3492>.

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

[RFC6760] Cheshire, S. and M. Krochmal, "Requirements for a Protocol to Replace the AppleTalk Name Binding Protocol (NBP)", RFC 6760, DOI 10.17487/RFC6760, February 2013, <https://www.rfc-editor.org/info/rfc6760>.

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

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

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

[RFC8375] Pfister, P. and T. Lemon, "Special-Use Domain 'home.arpa.'", RFC 8375, DOI 10.17487/RFC8375, May 2018, <https://www.rfc-editor.org/info/rfc8375>.

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

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

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

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

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

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

Appendix A. Implementation Status

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

A.1. Already Implemented and Deployed

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

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

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

Domain enumeration and unicast querying have been used for several years at IETF meetings to make Terminal Room printers discoverable from outside the Terminal room. When an IETF attendee presses Cmd-P on a Mac, or selects AirPrint on an iPad or iPhone, and the Terminal room printers appear, that is because the client is sending unicast DNS queries to the IETF DNS servers. A 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 has an implementation that runs on any Unix/Linux. It has a RESTful interface for management and an experimental demo CLI and web interface.

Ted Lemon also has produced a portable implementation of Discovery Proxy, which is available in the mDNSResponder open source code.

The Long-Lived Query mechanism [LLQ] referred to in this specification exists and is deployed, but was not standardized by the IETF. The IETF has developed a superior Long-Lived Query mechanism called DNS Push Notifications [Push], which is built on DNS Stateful Operations [RFC8490]. The pragmatic short-term deployment approach is for vendors to produce Discovery Proxies that implement both the deployed Long-Lived Query mechanism [LLQ] (for today's clients) and the new DNS Push Notifications mechanism [Push] as the preferred long-term direction.

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.

Author's Address

Stuart Cheshire
Apple Inc.
One Apple Park Way
Cupertino, California 95014
USA

Phone: +1 (408) 996-1010
Email: cheshire@apple.com

Network Working Group C. Huitema
Internet-Draft Private Octopus Inc.
Intended status: Standards Track D. Kaiser
Expires: April 18, 2019 October 15, 2018

Device Pairing Using Short Authentication Strings
draft-ietf-dnssd-pairing-05

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

Copyright Notice

Table of Contents

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 is Section 3.

DNSSD privacy requirements are analyzed in [I-D.ietf-dnssd-prireq] and scaling considerations are reviewed in [I-D.ietf-dnssd-privacyscaling]. Further work on these two drafts may lead to reviewing the mechanism proposed here.

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 and 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:

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

messageType Set to "ServerRandom".

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

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

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

messageType Set to "ClientRandom".

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

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

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

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

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

Both client and server display the SAS as a 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

```
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

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

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

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

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

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

7.2. Informative References

[I-D.ietf-dnssd-pairing-info]
Kaiser, D. and C. Huitema, "Device Pairing Design Issues", draft-ietf-dnssd-pairing-info-01 (work in progress), April 2018.

[I-D.ietf-dnssd-prireq]
Huitema, C., "DNS-SD Privacy and Security Requirements", draft-ietf-dnssd-prireq-00 (work in progress), September 2018.

[I-D.ietf-dnssd-privacy]
Huitema, C. and D. Kaiser, "Privacy Extensions for DNS-SD", draft-ietf-dnssd-privacy-04 (work in progress), April 2018.

[I-D.ietf-dnssd-privacyscaling]
Huitema, C., "DNS-SD Privacy Scaling Tradeoffs", draft-ietf-dnssd-privacyscaling-00 (work in progress), September 2018.

[K17] Kaiser, D., "Efficient Privacy-Preserving Configurationless Service Discovery Supporting Multi-Link Networks", 2017, <http://nbn-resolving.de/urn:nbn:de:bsz:352-0-422757>.

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

Network Working Group D. Kaiser
Internet-Draft
Intended status: Informational C. Huitema
Expires: April 26, 2019 Private Octopus Inc.
October 23, 2018

Device Pairing Design Issues
draft-ietf-dnssd-pairing-info-02

Abstract

This document discusses issues and problems occuring 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

Copyright Notice

Table of Contents

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]. The requirement of a pairing system for private discovery are analyzed in part in [I-D.ietf-dnssd-prireq].

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 systems emits a sequence of sounds or ultrasounds that is picked by the microphone of the other system.

A common problem with these solutions is that they require special capabilities that may not be present in every device. Another problem is that they are often one-way channels.

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 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^xA -->
                       <-- g^xB
nA -->
                          <-- nB
Computes              Computes
s = g^xAxB            s = g^xAxB
h = hash(s|nA|nB)     h = hash(s|nA|nB)
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^xA -->
                             g^xA'-->
                                             <-- g^xB
                          <--g^xB'
       nA -->
                             nA -->
                                               <-- nB
                           Picks nB'
                           smartly
                          <--nB'
       Computes                              Computes
       s' = g^xAxB'                            s" = g^xA'xB
       h' = hash(s'|nA|nB')                     h" = hash(s"|nA|nB)
       Displays short                        Displays short
       version of h'                         version of h"
```

In order to pick a nonce nB' that circumvents this naive security measure, Eve runs the following algorithm:

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

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 -->
                          <-- g^xB

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

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

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

There is still a theoretical problem, if Eve has somehow managed to "crack" the hash function. We 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 likelyhood 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:

o 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.

o 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:

 mmmmmmm  m  m mmmmmmm
 # mmm # ## "m # mmm #
 # ### # m" #" # ### #
 #mmmmm# # m m #mmmmm#
 mm m  mm"## m mmm mm
 " ##"mm m"# ####"m""#
 #"mmm mm# m"# ""m" "m
 mmmmmmm #mmm###mm# m
 # mmm #  m "mm " "  "
 # ### # " m #  "## "#
 #mmmmm# ### 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

 mmmmmmm   mmm mmmmmmm
 # mmm # ""#m# # mmm #
 # ### # "#  # # ### #
 #mmmmm# # m"m #mmmmm#
 mmmmm mmm" m m m m m
  #"m mmm#"#"#"#m m#m
 ""mmmmm"m#""#""m #  m
 mmmmmmm # "m"m "m"#"m
 # mmm # mmmm m "# #"
 # ### # #mm"#"#m "
 #mmmmm# #mm"#""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

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

[I-D.ietf-dnssd-pairing]
Huitema, C. and D. Kaiser, "Device Pairing Using Short Authentication Strings", draft-ietf-dnssd-pairing-04 (work in progress), April 2018.

[I-D.ietf-dnssd-prireq]
Huitema, C., "DNS-SD Privacy and Security Requirements", draft-ietf-dnssd-prireq-00 (work in progress), September 2018.

[I-D.ietf-dnssd-privacy]
Huitema, C. and D. Kaiser, "Privacy Extensions for DNS-SD", draft-ietf-dnssd-privacy-04 (work in progress), April 2018.

[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.

[K17] Kaiser, D., "Efficient Privacy-Preserving Configurationless Service Discovery Supporting Multi-Link Networks", 2017, <http://nbn-resolving.de/urn:nbn:de:bsz:352-0-422757>.

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

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

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

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

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

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

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

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

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

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

Authors' Addresses

Daniel Kaiser
Esch-sur-Alzette 4360
Luxembourg

Email: daniel@kais3r.de

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

Email: huitema@huitema.net

Network Working Group C. Huitema
Internet-Draft Private Octopus Inc.
Intended status: Standards Track D. Kaiser
Expires: April 18, 2019 University of Konstanz
October 15, 2018

Privacy Extensions for DNS-SD
draft-ietf-dnssd-privacy-05

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.

Revisions of this draft are currently considered in the DNSSD working group.

Table of Contents

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 [I-D.ietf-dnssd-prireq], 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 2, and describe the proposed solution in Section 3.

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.

The solution presented here relies on 1-1 pairings between clients and servers. Discussions during the IETF 101 in London showed that this requirement of a full mesh of pairings poses some scalability issues, as explained in [I-D.ietf-dnssd-privacyscaling]. The next revision of this draft may propose a different mechanism.

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. 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

- o 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.

- o 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.

2.1. 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.

2.2. 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.

2.2.1. 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.

2.2.2. 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.

2.2.3. 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 3.5 but in the absence of the proper pairwise shared key, the connection will fail. This will not create an actual error, but the probability of such events should be kept low.

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

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

Table 1

The table shows that for a proof, 24 bits would be too short. 32 bits might be long enough, but the BASE64 encoding requires padding if the input is not an even multiple of 24 bits, and BASE32 requires padding if the input is not a multiple of 40 bits. Given that, the desirable proof lengths are thus 48 bits if using BASE64, or 56 bits if using BASE32. The resulting instance name will be either 12 characters long with BASE64, allowing 54 advertisements in an 1500 byte mDNS message, or 16 characters long with BASE32, allowing 47 advertisements per message.

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

2.2.4. 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.

2.3. 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:

```
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 2.2, by concatenating a nonce and the hash of the nonce with the shared secret.

2.3.1. 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:

- o 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.

- o 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.

- o 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.

2.4. 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.

2.5. 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.

3. 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.

3.1. 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 2.5. 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.

3.2. 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.

3.3. 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 3.2). 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.

3.3.1. 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 3.4, 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.

3.4. 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].

3.5. 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 3.6. 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.

3.6. 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 3.5.

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 3.4.

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 3.4.

3.7. 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.

4. 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.

4.1. 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 3.5. 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].

4.2. Denial of Discovery of the Private Discovery Service

The algorithm described in Section 3.5 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 3.5. 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.

4.3. 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 3.4 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 3.5.

4.4. 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.

4.5. 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 3.3.1 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.

4.6. 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.

4.7. 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.

5. IANA Considerations

This draft does not require any IANA action.

6. 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.

7. References

7.1. Normative References

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

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

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

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

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

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

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

7.2. Informative References

[I-D.ietf-dnssd-pairing]
Huitema, C. and D. Kaiser, "Device Pairing Using Short Authentication Strings", draft-ietf-dnssd-pairing-04 (work in progress), April 2018.

[I-D.ietf-dnssd-prireq]
Huitema, C., "DNS-SD Privacy and Security Requirements", draft-ietf-dnssd-prireq-00 (work in progress), September 2018.

[I-D.ietf-dnssd-privacyscaling]
Huitema, C., "DNS-SD Privacy Scaling Tradeoffs", draft-ietf-dnssd-privacyscaling-00 (work in progress), September 2018.

[I-D.ietf-dnssd-push]
Pusateri, T. and S. Cheshire, "DNS Push Notifications", draft-ietf-dnssd-push-15 (work in progress), September 2018.

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-28 (work in progress), March 2018.

[K17] Kaiser, D., "Efficient Privacy-Preserving Configurationless Service Discovery Supporting Multi-Link Networks", 2017, <http://nbn-resolving.de/urn:nbn:de:bsz:352-0-422757>.

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

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

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

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

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

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

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

[RFC8094] Reddy, T., Wing, D., and P. Patil, "DNS over Datagram Transport Layer Security (DTLS)", RFC 8094, DOI 10.17487/RFC8094, February 2017, <https://www.rfc-editor.org/info/rfc8094>.

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

Authors' Addresses

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

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

Daniel Kaiser
University of Konstanz
Konstanz 78457
Germany

Email: daniel.kaiser@uni-konstanz.de

Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: February 8, 2020

T. Pusateri
Unaffiliated
S. Cheshire
Apple Inc.

August 7, 2019

DNS Push Notifications
draft-ietf-dnssd-push-24

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.

Table of Contents

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 BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here. These words may also appear in this document in lower case as plain English words, absent their normative meanings.

1.2. Fatal Errors

Certain invalid situations are described in this specification, like a server sending a Push Notification subscription request to a client, or a client sending a Push Notification response to a server. These should never occur with a correctly implemented client and server, and if they do occur then they indicate a serious implementation error. In these extreme cases there is no reasonable expectation of a graceful recovery, and the recipient detecting the error should respond by unilaterally aborting the session without regard for data loss. Such cases are addressed by having an engineer investigate the cause of the failure and fixing the problem in the software.

Where this specification says "forcibly abort", it means sending a TCP RST to terminate the TCP connection, and the TLS session running over that TCP connection. In the BSD Sockets API, this is achieved by setting the SO_LINGER option to zero before closing the socket.

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 address, 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 [RFC6281] service 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) [RFC8490] 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 depending on the gateway to support NAT Port Mapping Protocol (NAT-PMP) [RFC6886] or Port Control Protocol (PCP) [RFC6887], or resorting to excessive keepalive traffic.

3. Overview

A DNS Push Notification client subscribes for Push Notifications for a particular RRset by connecting to the appropriate Push Notification server for that RRset, and sending DSO 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 DNS zone is any server capable of generating the correct change notifications for a name. It may be a primary, secondary, or stealth name server [RFC7719].

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 DSO session to that server for both DNS Updates and DNS Push Notification Subscriptions. 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. Supporting DNS Updates and DNS Push Notifications on the same server is OPTIONAL. A DNS Push Notification server is not required to support DNS Update.

Standard DNS Queries MAY be sent over a DNS Push Notification (i.e., DSO) session. For any zone for which the server is authoritative, it MUST respond authoritatively for queries for names falling within that zone (e.g., the "_dns-push-tls._tcp.<zone>" SRV record) both for normal DNS queries and for DNS Push Notification subscriptions. For names for which the server is acting as a recursive resolver, e.g., when the server is the local recursive resolver, for any query for which it supports DNS Push Notification subscriptions, it MUST also support standard queries.

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). In the case of a device like a smartphone which, after some period of inactivity, goes to sleep or otherwise darkens its screen, it should cancel its subscriptions when darkening the screen (since the user cannot see any changes on the display anyway) and reinstate its subscriptions when re-awakening from display sleep.

(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 [RFC8490], a client must not keep a DSO session to a server open indefinitely if it has no subscriptions (or other operations) active on that session. A client MAY close a DSO session immediately it becomes idle, and then if needed in the future, open a new session when required. Alternatively, a client MAY speculatively keep an idle DSO 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) [RFC8490].

Note that a DSO session which has an active DNS Push Notification subscription is not considered idle, even if there is no traffic flowing for an extended period of time. In this case the DSO inactivity timeout does not apply, because the session is not inactive, but the keepalive interval does still apply, to ensure generation of sufficient messages to maintain state in middleboxes (such at NAT gateways or firewalls) and for the client and server to periodically verify that they still have connectivity to each other. This is described in Section 6.2 of the DSO specification [RFC8490].

4. 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.

5. 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 [RFC8490] running over TLS over TCP [RFC7858].

Connection setup over TCP ensures return reachability and alleviates concerns of state overload at the server which is a potential problem with connectionless protocols using spoofed source addresses. 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], RACK: a time-based fast loss detection algorithm for TCP [I-D.ietf-tcpm-rack], and so on.

Transport Layer Security (TLS) [RFC8446] is well understood, and used by many application-layer protocols running over TCP. TLS 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.

6. Protocol Operation

The DNS Push Notification protocol is a session-oriented protocol, and makes use of DNS Stateful Operations (DSO) [RFC8490].

For details of the DSO message format refer to the DNS Stateful Operations specification [RFC8490]. Those details are not repeated here.

DNS Push Notification clients and servers MUST support DSO. A single server can support DNS Queries, DNS Updates, and DNS Push Notifications (using DSO) on the same TCP port.

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 and/or keepalive interval longer than the the 15-second default values, 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. Sending either a DSO Keepalive operation or a Push Notification subscription request over the TLS/TCP connection to the server signals the client's support of DSO and serves to establish a DSO session.

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 establishing a 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. This connection is made to TCP port 853, the default port for DNS-over-TLS [RFC7858]. If the request for a Push Notification subscription is successful, and the recursive resolver doesn't already have an active subscription for that name, type, and class, then the recursive resolver will make a corresponding Push Notification subscription on the client's behalf. Results received are relayed to the client. This is closely analogous to how a client sends a normal DNS query to its configured DNS recursive resolver which, if it doesn't already have appropriate answer(s) in its cache, issues an upstream query to satisfy the request.

In many contexts, the recursive resolver will be able to handle Push Notifications for all names that the client may need to follow. Use of VPN tunnels and split-view DNS can create some additional complexity in the client software here; the techniques to handle VPN tunnels and split-view DNS for DNS Push Notifications are the same as those already used to handle this for normal DNS queries.

If the recursive resolver does not support DNS over TLS, or supports DNS over TLS but is not listening on TCP port 853, or supports DNS over TLS on TCP port 853 but does not support DSO on that port, then the DSO Session session establishment will fail [RFC8490].

If the recursive resolver does support DSO but not Push Notification subscriptions, then it will return the DSO error code, DSOTYPENI (11).

In some cases, the recursive resolver may support DSO and Push Notification subscriptions, but may not be able to subscribe for Push Notifications for a particular name. In this case, the recursive resolver should return SERVFAIL to the client. This includes being unable to establish a connection to the zone's DNS Push Notification server or establishing a connection but receiving a non success response code. In some cases, where the client has a pre-established trust relationship with the owner of the zone (that is not handled via the usual mechanisms for VPN software) the client may handle these failures by contacting the zone's DNS Push server directly.

In any of the cases described above where the client fails to establish a DNS Push Notification subscription via its configured recursive resolver, 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.

The discovery 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.headoffice.example.com (to discover Internet Printing Protocol (IPP) printers [RFC8010] [RFC8011] being advertised in the head office of Example Company.). The client begins by sending an SOA query for _ipp._tcp.headoffice.example.com to the local recursive resolver. The goal is to determine the server authoritative for the name _ipp._tcp.headoffice.example.com. The closest enclosing DNS zone containing the name _ipp._tcp.headoffice.example.com could be example.com, or headoffice.example.com, or _tcp.headoffice.example.com, or even _ipp._tcp.headoffice.example.com. The client does not know in advance where the closest enclosing zone cut occurs, which is why it uses the iterative 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 language is not placing any new requirements on DNS recursive resolvers. This text merely describes 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 would normally include the SOA record for the closest enclosing 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 language is not placing any new requirements on DNS recursive resolvers. This text merely describes 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 two labels in it, the client sends an SOA query for that new name, and processing continues at step 2 above, repeating the iterative search until either an SOA is received, or the query name consists of a single label, i.e., a Top Level Domain (TLD). In the case of a single-label (TLD), 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 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 below in Section 7.2 and in the specification for using DANE TLSA Records with SRV Records [RFC7673], if applicable.

7. More than one SRV record may be returned. In this case, the "priority" and "weight" values in the returned SRV records are used to determine the order in which to contact the servers for subscription requests. As described in the SRV specification [RFC2782], the server with the lowest "priority" is first contacted. If more than one server has the same "priority", the "weight" indicates the weighted probability that the client should contact that server. Higher weights have higher probabilities of being selected. If a server is not 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 are 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 to the server. A SUBSCRIBE request is encoded in a DSO message [RFC8490]. This specification defines a primary DSO TLV for DNS Push Notification SUBSCRIBE Requests (tentatively DSO Type Code 0x40).

DSO messages with the SUBSCRIBE TLV as the Primary TLV are permitted in TLS early data, provided that the precautions described in Section 7.3 are followed.

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 MUST forcibly abort the connection immediately.

6.2.1. SUBSCRIBE Request

A SUBSCRIBE request begins with the standard DSO 12-byte header [RFC8490], followed by the SUBSCRIBE primary 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 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 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 [RFC8490]. The DNS OPCODE field contains the OPCODE value for DNS Stateful Operations (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.

```
                                1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  \
|                  MESSAGE ID                   |   \
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
|QR| OPCODE(6) |           Z          |  RCODE  |    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
|             QDCOUNT (MUST BE ZERO)            |    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+     > HEADER
|             ANCOUNT (MUST BE ZERO)            |    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
|             NSCOUNT (MUST BE ZERO)            |    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
|             ARCOUNT (MUST BE ZERO)            |   /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  /
|    DSO-TYPE = SUBSCRIBE (tentatively 0x40)    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|   DSO-LENGTH (number of octets in DSO-DATA)   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  \
|                                               |   \
\                     NAME                      \    |
\                                               \    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+     > DSO-DATA
|                     TYPE                      |    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
|                     CLASS                     |   /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  /
```

Figure 1: SUBSCRIBE Request

The DSO-DATA for a SUBSCRIBE request MUST contain exactly one NAME, TYPE, and CLASS. Since SUBSCRIBE requests are sent over TCP, multiple SUBSCRIBE DSO 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 a fatal error and the server MUST forcibly abort the connection immediately.

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 no other records with the same owner name.

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 NOT return NXDOMAIN in this case. 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 = 255 (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 begins with the standard DSO 12-byte header [RFC8490]. The QR bit in the header is set indicating it is a response. The header MAY be followed by one or more optional TLVs, such as a Retry Delay TLV.

The MESSAGE ID field MUST echo the value given in the MESSAGE 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.

A client MUST NOT send a SUBSCRIBE response. If a client does send a SUBSCRIBE message, with the QR bit set indicating that it is a response, this is a fatal error and the server MUST forcibly abort the connection immediately.

```
                                 1  1  1  1  1  1
   0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  \
 |                  MESSAGE ID                   |   \
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
 |QR| OPCODE(6) |         Z          |   RCODE   |    |
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
 |             QDCOUNT (MUST BE ZERO)            |    |
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+     > HEADER
 |             ANCOUNT (MUST BE ZERO)            |    |
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
 |             NSCOUNT (MUST BE ZERO)            |    |
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
 |             ARCOUNT (MUST BE ZERO)            |   /
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  /
```

Figure 2: SUBSCRIBE Response Message

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

Mnemonic	Value	Description
NOERROR	0	SUBSCRIBE successful.
FORMERR	1	Server failed to process request due to a malformed request.
SERVFAIL	2	Server failed to process request due to a problem with the server.
NOTIMP	4	Server does not implement DSO.
REFUSED	5	Server refuses to process request for policy or security reasons.
NOTAUTH	9	Server is not authoritative for the requested name.
DSOTYPENI	11	SUBSCRIBE operation not supported.

Table 1: SUBSCRIBE Response codes

This document specifies only these RCODE values for SUBSCRIBE Responses. Servers sending SUBSCRIBE Responses SHOULD use one of these values. Note that NXDOMAIN is not a valid RCODE in response to a SUBSCRIBE Request. 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 other 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 to this server right away. If multiple SRV records were returned as described in Section 6.1, Paragraph 7, a subsequent server can be tried immediately.

If the client has other successful subscriptions to this server, these subscriptions remain even though additional subscriptions may be refused. Neither the client nor the server are required to close the connection, although, either end may choose to do so.

If the server sends a nonzero RCODE then it SHOULD append a Retry Delay TLV [RFC8490] 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 DNS Stateful Operations [RFC8490], 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.

If the server being queried is listed in a "_dns-push-tls._tcp.<zone>" SRV record for the zone, then this is a misconfiguration, since this server is being advertised as supporting DNS Push Notifications for this zone, but the server itself is not currently configured to perform that task. 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.

If the server being queried is listed in a "_dns-push-tls._tcp.<zone>" SRV record for the zone, then this is a misconfiguration, since this server is being advertised as supporting DNS Push Notifications for this zone, but the server itself is not currently configured to perform that task. 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 (DSOTYPENI), which occurs on a server that implements DSO but 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 [RFC8490]. 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 already 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 unidirectional message begins with the standard DSO 12-byte header [RFC8490], followed by the PUSH primary TLV. A PUSH message is illustrated in Figure 3.

In accordance with the definition of DSO unidirectional messages, 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 [RFC8490]. The DNS OPCODE field contains the OPCODE value for DNS Stateful Operations (6). The four count fields MUST be zero, and the corresponding four sections MUST be empty (i.e., absent).

A client MUST NOT send a PUSH message. If a client does send a PUSH message, or a PUSH message is sent with the QR bit set indicating that it is a response, this is a fatal error and the receiver MUST forcibly abort the connection immediately.

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 change notifications. A PUSH Message MUST contain at least one change notification. If a PUSH Message is received that contains no change notifications, this is a fatal error, and the client MUST forcibly abort the connection immediately.

The change notification records are formatted similarly to how DNS Resource Records are conventionally expressed in DNS messages, as illustrated in Figure 3, and are interpreted as described below.

The TTL field holds an unsigned 32-bit integer [RFC2181]. If the TTL is in the range 0 to 2,147,483,647 seconds (2^31 - 1, or 0x7FFFFFFF), then a new DNS Resource Record with the given name, type, class and

RDATA is added. A TTL of 0 means that this record should be retained for as long as the subscription is active, and should be discarded immediately the moment the subscription is cancelled.

If the TTL has the value 0xFFFFFFFF, then the DNS Resource Record with the given name, type, class and RDATA is removed.

If the TTL has the value 0xFFFFFFFE, then this is a 'collective' remove notification. For collective remove notifications RDLEN MUST be zero and consequently the RDATA MUST be empty. If a change notification is received where TTL = 0xFFFFFFFE and RDLEN is not zero, this is a fatal error, and the client MUST forcibly abort the connection immediately.

There are three types of collective remove notification:

For collective remove notifications, if CLASS is not 255 (ANY) and TYPE is not 255 (ANY) then for the given name this deletes all records of the specified type in the specified class.

For collective remove notifications, if CLASS is not 255 (ANY) and TYPE is 255 (ANY) then for the given name this deletes all records of all types in the specified class.

For collective remove notifications, if CLASS is 255 (ANY), then for the given name this deletes all records of all types in all classes. In this case TYPE MUST be set to zero on transmission, and MUST be silently ignored on reception.

Summary of change notification types:

```
   Delete all RRsets from a name, in all classes
   TTL = 0xFFFFFFFE, RDLEN = 0, CLASS = 255 (ANY)

   Delete all RRsets from a name, in given class:
   TTL = 0xFFFFFFFE, RDLEN = 0, CLASS gives class, TYPE = 255 (ANY)

   Delete specified RRset from a name, in given class:
   TTL = 0xFFFFFFFE, RDLEN = 0
   CLASS and TYPE specify the RRset being deleted

   Delete an individual RR from a name:
   TTL = 0xFFFFFFFF
   CLASS, TYPE, RDLEN and RDATA specify the RR being deleted.

   Add individual RR to a name
   TTL >= 0 and TTL <= 0x7FFFFFFF
   CLASS, TYPE, RDLEN, RDATA and TTL specify the RR being added.
```

Note that it is valid for the RDATA of an added or removed DNS Resource Record to be empty (zero length). For example, an Address Prefix List Resource Record [RFC3123] may have empty RDATA. Therefore, a change notification with RDLEN = 0 does not automatically indicate a remove notification. If RDLEN = 0 and TTL is the in the range 0 - 0x7FFFFFFF, this change notification signals the addition of a record with the given name, type, class, and empty RDATA. If RDLEN = 0 and TTL = 0xFFFFFFFF, this change notification signals the removal specifically of that single record with the given name, type, class, and empty RDATA.

If the TTL is any value other than 0xFFFFFFFF, 0xFFFFFFFE, or a value in the range 0 - 0x7FFFFFFF, then the receiver SHOULD silently ignore this particular change notification record. The connection is not terminated and other valid change notification records within this PUSH message are processed as usual.

For efficiency, when generating a PUSH message, a server SHOULD include as many change notifications as it has immediately available to send, rather than sending each change notification as a separate DSO message. Once it has exhausted the list of change notifications immediately available to send, a server SHOULD then send the PUSH message immediately, rather than waiting to see if additional change notifications become available.

For efficiency, when generating a PUSH message, a server SHOULD use standard DNS name compression, with offsets relative to the beginning of the DNS message [RFC1035]. When multiple change notifications in a single PUSH message have the same owner name, this name compression can yield significant savings. Name compression should be performed as specified in Section 18.14 of the Multicast DNS specification [RFC6762], namely, owner names should always be compressed, and names appearing within RDATA should be compressed for only the RR types listed below:

NS, CNAME, PTR, DNAME, SOA, MX, AFSDB, RT, KX, RP, PX, SRV, NSEC

Servers may generate PUSH messages up to a maximum DNS message length of 16,382 bytes, counting from the start of the DSO 12-byte header. Including the two-byte length prefix that is used to frame DNS over a byte stream like TLS, this makes a total of 16,384 bytes. Servers MUST NOT generate PUSH messages larger than this. Where the immediately available change notifications are sufficient to exceed a DNS message length of 16,382 bytes, the change notifications MUST be communicated in separate PUSH messages of up to 16,382 bytes each. DNS name compression becomes less effective for messages larger than 16,384 bytes, so little efficiency benefit is gained by sending messages larger than this.

If a client receives a PUSH message with a DNS message length larger than 16,382 bytes, this is a fatal error, and the client MUST forcibly abort the connection immediately.

```
                                1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  \
|          MESSAGE ID (MUST BE ZERO)            |   \
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
|QR| OPCODE(6) |         Z          |   RCODE   |    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
|             QDCOUNT (MUST BE ZERO)            |    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+     > HEADER
|             ANCOUNT (MUST BE ZERO)            |    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
|             NSCOUNT (MUST BE ZERO)            |    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
|             ARCOUNT (MUST BE ZERO)            |   /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  /
|      DSO-TYPE = PUSH (tentatively 0x41)       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|   DSO-LENGTH (number of octets in DSO-DATA)   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  \
\                     NAME                      \   \
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
|                     TYPE                      |    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
|                     CLASS                     |    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
|                      TTL                      |    |
|      (32-bit unsigned big-endian integer)     |     > DSO-DATA
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
|  RDLEN (16-bit unsigned big-endian integer)   |    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
\          RDATA (sized as necessary)           \    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
:     NAME, TYPE, CLASS, TTL, RDLEN, RDATA      :    |
:             Repeated As Necessary             :   /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  /
```

Figure 3: 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 a 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 = 255 (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. While the subscription is active, the TTL is not decremented, because a change to the TTL would produce a new update. 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 for records covered by the subscription 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 as a DSO unidirectional message [RFC8490]. This specification defines a primary unidirectional DSO TLV for DNS Push Notification UNSUBSCRIBE Messages (tentatively DSO Type Code 0x42).

A server MUST NOT send an UNSUBSCRIBE message. If a server does send an UNSUBSCRIBE message over a DSO session initiated by a client, or an UNSUBSCRIBE message is sent with the QR bit set indicating that it is a response, this is a fatal error and the receiver MUST forcibly abort the connection immediately.

6.4.1. UNSUBSCRIBE Message

An UNSUBSCRIBE unidirectional message begins with the standard DSO 12-byte header [RFC8490], followed by the UNSUBSCRIBE primary TLV. An UNSUBSCRIBE message is illustrated in Figure 4.

In accordance with the definition of DSO unidirectional messages, the MESSAGE ID field MUST be zero. There is no server response to an UNSUBSCRIBE message.

The other header fields MUST be set as described in the DSO specification [RFC8490]. The DNS OPCODE field contains the OPCODE value for DNS Stateful Operations (6). The four count fields MUST be zero, and the corresponding four sections MUST be empty (i.e., absent).

The DSO-TYPE is UNSUBSCRIBE (tentatively 0x42).

The DSO-LENGTH field contains the value 2, the length of the 2-octet MESSAGE ID contained in the DSO-DATA.

The DSO-DATA contains the value given in the MESSAGE ID field of an active SUBSCRIBE request. This is how the server knows which SUBSCRIBE request is being cancelled. After receipt of the UNSUBSCRIBE message, the SUBSCRIBE request is no longer active.

It is allowable for the client to issue an UNSUBSCRIBE message 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 message.

Consequently, it is possible for a server to receive an UNSUBSCRIBE message that does not match any currently active subscription. This can occur when a client sends a SUBSCRIBE request, which subsequently fails and returns an error code, but the client sent an UNSUBSCRIBE message before it became aware that the SUBSCRIBE request had failed. Because of this, servers MUST silently ignore UNSUBSCRIBE messages that do not match any currently active subscription.

```
                                1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  \
|          MESSAGE ID (MUST BE ZERO)            |   \
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
|QR| OPCODE(6) |         Z          |   RCODE   |    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
|            QDCOUNT (MUST BE ZERO)             |    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+     > HEADER
|            ANCOUNT (MUST BE ZERO)             |    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
|            NSCOUNT (MUST BE ZERO)             |    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
|            ARCOUNT (MUST BE ZERO)             |   /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  /
|   DSO-TYPE = UNSUBSCRIBE (tentatively 0x42)   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                 DSO-LENGTH (2)                |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  \
|             SUBSCRIBE MESSAGE ID              |   > DSO-DATA
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  /
```

Figure 4: UNSUBSCRIBE Message

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 believes 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 message 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 queries to ascertain whether the target device is still present. How the Discovery Proxy causes these new Multicast DNS queries to be issued depends on the details of the underlying Multicast DNS implementation being used. For example, a Discovery Proxy built on Apple's dns_sd.h API responds to a DNS Push Notification RECONFIRM message by calling the underlying API's DNSServiceReconfirmRecord() routine.

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 use of 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 message, 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.

A server MUST NOT send a RECONFIRM message. If a server does send a RECONFIRM message over a DSO session initiated by a client, or a RECONFIRM message is sent with the QR bit set indicating that it is a response, this is a fatal error and the receiver MUST forcibly abort the connection immediately.

6.5.1. RECONFIRM Message

A RECONFIRM unidirectional message begins with the standard DSO 12-byte header [RFC8490], followed by the RECONFIRM primary TLV. A RECONFIRM message is illustrated in Figure 5.

In accordance with the definition of DSO unidirectional messages, the MESSAGE ID field MUST be zero. There is no server response to a RECONFIRM message.

The other header fields MUST be set as described in the DSO specification [RFC8490]. The DNS OPCODE field contains the OPCODE value for DNS Stateful Operations (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 message MUST contain exactly one record. The DSO-DATA for a RECONFIRM message has no count field to specify more than one record. Since RECONFIRM messages are sent over TCP, multiple RECONFIRM 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 no other records with the same owner name.

Note that there is no RDLEN field, since the length of the RDATA can be inferred from DSO-LENGTH, so an additional RDLEN field would be redundant.

```
                                  1  1  1  1  1  1
    0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  \
  |                  MESSAGE ID                   |   \
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
  |QR| OPCODE(6) |         Z          |   RCODE   |    |
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
  |             QDCOUNT (MUST BE ZERO)            |    |
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+     > HEADER
  |             ANCOUNT (MUST BE ZERO)            |    |
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
  |             NSCOUNT (MUST BE ZERO)            |    |
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
  |             ARCOUNT (MUST BE ZERO)            |   /
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  /
  |    DSO-TYPE = RECONFIRM (tentatively 0x43)    |
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
  |   DSO-LENGTH (number of octets in DSO-DATA)   |
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  \
  \                     NAME                      \   \
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
  |                     TYPE                      |    |
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+     > DSO-DATA
  |                     CLASS                     |    |
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+    |
  \                     RDATA                     \   /
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  /
```

Figure 5: RECONFIRM Message

6.6. DNS Stateful Operations TLV Context Summary

This document defines four new DSO TLVs. As suggested in Section 8.2 of the DNS Stateful Operations specification [RFC8490], the valid contexts of these new TLV types are summarized below.

The client TLV contexts are:

C-P: Client request message, primary TLV
C-U: Client unidirectional message, primary TLV
C-A: Client request or unidirectional message, additional TLV
CRP: Response back to client, primary TLV
CRA: Response back to client, additional TLV

TLV Type	C-P	C-U	C-A	CRP	CRA
SUBSCRIBE	X				
PUSH					
UNSUBSCRIBE		X			
RECONFIRM		X			

Table 2: DSO TLV Client Context Summary

The server TLV contexts are:

S-P: Server request message, primary TLV
S-U: Server unidirectional message, primary TLV
S-A: Server request or unidirectional message, additional TLV
SRP: Response back to server, primary TLV
SRA: Response back to server, additional TLV

TLV Type	S-P	S-U	S-A	SRP	SRA
SUBSCRIBE					
PUSH		X			
UNSUBSCRIBE					
RECONFIRM					

Table 3: DSO TLV Server Context Summary

6.7. 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 [RFC8490].

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 should perform an orderly close of the TLS session. Typical APIs will provide a session close method that will send a TLS close_notify alert (see Section 6.1 of the TLS 1.3 specification [RFC8446]). This instructs the recipient that the sender will not send any more data over the session. After sending the TLS close_notify alert the client MUST gracefully close the underlying connection using a TCP FIN, so that the TLS close_notify is reliably delivered. The mechanisms for gracefully closing a TCP connection with a TCP FIN vary depending on the networking API. For example, 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.

If the session is forcibly closed at the TCP level by sending a RST from either end of the connection, data may be lost.

6.8. Client Fallback to Polling

There are cases where a client may exhaust all avenues for establishing a DNS Push Notification subscription without success. This can happen if the client's configured recursive resolver does not support DNS over TLS, or supports DNS over TLS but is not listening on TCP port 853, or supports DNS over TLS on TCP port 853 but does not support DSO on that port, or for some other reason is unable to provide a DNS Push Notification subscription. In this case the client will attempt to communicate directly with an appropriate server, and it may be that the zone apex discovery fails, or there is no "_dns-push-tls._tcp.<zone>" SRV record, or server indicated in the SRV record is misconfigured, or is unresponsive for some other reason.

Regardless of the reason for the failure, after being unable to establish the desired DNS Push Notification subscription, it is likely that the client will still wish to know the answer it seeks, even if that answer cannot be obtained with the timely change notifications provided by DNS Push Notifications. In such cases it is likely that the client will obtain the answer it seeks via a conventional DNS query instead, repeated at some interval to detect when the answer RRset changes.

In the case where a client responds to its failure to establish a DNS Push Notification subscription by falling back to polling with conventional DNS queries instead, the polling rate should be controlled to avoid placing excessive burden on the server. The interval between successive DNS queries for the same name, type and class SHOULD be at least the minimum of: 900 seconds (15 minutes), or two seconds more than the TTL of the answer RRset.

The reason that for TTLs shorter than 898 seconds the query should not be reissued until two seconds *after* the answer RRset has expired is to ensure that the answer RRset has also expired from the cache on the client's configured recursive resolver. Otherwise (particularly if the clocks on the client and the recursive resolver do not run at precisely the same rate) there's a risk of a race condition where the client queries its configured recursive resolver just as the answer RRset has one second remaining in the recursive resolver's cache. The client would then receive a reply telling it that the answer RRset has one second remaining, and then the client would then re-query the recursive resolver again one second later when the answer RRset actually expires, and only then would the recursive resolver issue a new query to fetch new fresh data from the authoritative server. Waiting until the answer RRset has definitely expired from the the cache on the client's configured recursive resolver avoids this race condition and unnecessary additional queries it causes.

Each time a client is about to reissue its query to discover changes to the answer RRset, it should first make a new attempt to establish a DNS Push Notification subscription, using previously cached DNS answers as appropriate. After a temporary misconfiguration has been remedied, this allows a client that is polling to return to using DNS Push Notifications for asynchronous notification of changes.

7. Security Considerations

The Strict Privacy Usage Profile for DNS over TLS is REQUIRED for DNS Push Notifications [RFC8310]. Cleartext connections for DNS Push Notifications are not permissible. Since this is a new protocol, transition mechanisms from the Opportunistic Privacy profile are unnecessary.

Also, see Section 9 of the DNS over (D)TLS Usage Profiles document [RFC8310] for additional recommendations for various versions of TLS usage.

As a consequence of requiring TLS, client certificate authentication and verification may also be enforced by the server for stronger client-server security or end-to-end security. However, recommendations for security in particular deployment scenarios are outside the scope of this document.

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.

Anti-replay protection: TLS provides for the detection of and prevention against messages sent previously over a TLS connection (such as DNS Push Notifications). Prior messages cannot be re-sent at a later time as a form of a man-in-the-middle attack.

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 useable, an SNI extension must be used for the target name to ensure the client is connecting to the server it has authenticated. If the target name does not have a usable TLSA record, then the use of the SNI extension is optional. See Usage Profiles for DNS over TLS and DNS over DTLS [RFC8310] for more information on authenticating domain names.

7.3. TLS Early Data

DSO messages with the SUBSCRIBE TLV as the Primary TLV are permitted in TLS early data. Using TLS early data can save one network round trip, and can result in the client obtaining results faster.

However, there are some factors to consider before using TLS early data.

TLS Early Data is not forward secret. In cases where forward secrecy of DNS Push Notification subscriptions is required, the client should not use TLS Early Data.

With TLS early data there are no guarantees of non-replay between connections. If packets are duplicated and delayed in the network, the later arrivals could be mistaken for new subscription requests. Generally this is not a major concern, since the amount of state generated on the server for these spurious subscriptions is small and short-lived, since the TCP connection will not complete the three-way handshake. Servers MAY choose to implement rate-limiting measures that are activated when the server detects an excessive number of spurious subscription requests.

For further guidance please see Section 2.3, Section 8, and Appendix E.5 of the TLS 1.3 specification [RFC8446].

7.4. TLS Session Resumption

TLS Session Resumption is permissible on DNS Push Notification servers. The server may keep TLS state with Session IDs [RFC8446] or operate in stateless mode by sending a Session Ticket [RFC5077] to the client for it to store. However, closing the TLS connection terminates the DSO session. 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 may allow a 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 a new service name to be published in the IANA Registry Service Types [RFC6335][ST] that is only applicable for the TCP protocol.

Name	Port	Value	Definition
DNS Push Notification Service Type	None	"_dns-push-tls._tcp"	Section 6.1

Table 4: IANA Service Type Assignments

This document also defines four new DNS Stateful Operation TLV types to be recorded in the IANA DSO Type Code Registry.

Name	Value	Early Data	Status	Definition
SUBSCRIBE	TBA (0x40)	OK	Standards Track	Section 6.2
PUSH	TBA (0x41)	NO	Standards Track	Section 6.3
UNSUBSCRIBE	TBA (0x42)	NO	Standards Track	Section 6.4
RECONFIRM	TBA (0x43)	NO	Standards Track	Section 6.5

Table 5: IANA DSO TLV Type Code Assignments

This document defines no new DNS OPCODEs or RCODEs.

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, Sara Dickinson, Mark Delany, Ralph Droms, Jan Komissar, Eric Rescorla, Michael Richardson, David Schinazi, Manju Shankar Rao, Robert Sparks, Markus Stenberg, Andrew Sullivan, Michael Sweet, Dave Thaler, Brian Trammell, Bernie Volz, Eric Vyncke, Christopher Wood, Liang Xia, and Soraia Zlatkovic. Ted Lemon provided clarifying text that was greatly appreciated.

10. References

10.1. Normative References

[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>.

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

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

[RFC1123] Braden, R., Ed., "Requirements for Internet Hosts - Application and Support", STD 3, RFC 1123, DOI 10.17487/RFC1123, October 1989, <https://www.rfc-editor.org/info/rfc1123>.

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

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

[RFC2181] Elz, R. and R. Bush, "Clarifications to the DNS Specification", RFC 2181, DOI 10.17487/RFC2181, July 1997, <https://www.rfc-editor.org/info/rfc2181>.

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

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC6335] Cotton, M., Eggert, L., Touch, J., Westerlund, M., and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry", BCP 165, RFC 6335, DOI 10.17487/RFC6335, August 2011, <https://www.rfc-editor.org/info/rfc6335>.

[RFC6895] Eastlake 3rd, D., "Domain Name System (DNS) IANA Considerations", BCP 42, RFC 6895, DOI 10.17487/RFC6895, April 2013, <https://www.rfc-editor.org/info/rfc6895>.

[RFC7673] Finch, T., Miller, M., and P. Saint-Andre, "Using DNS-Based Authentication of Named Entities (DANE) TLSA Records with SRV Records", RFC 7673, DOI 10.17487/RFC7673, October 2015, <https://www.rfc-editor.org/info/rfc7673>.

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

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC8490] Bellis, R., Cheshire, S., Dickinson, J., Dickinson, S., Lemon, T., and T. Pusateri, "DNS Stateful Operations", RFC 8490, DOI 10.17487/RFC8490, March 2019, <https://www.rfc-editor.org/info/rfc8490>.

[ST] "Service Name and Transport Protocol Port Number Registry", <http://www.iana.org/assignments/service-names-port-numbers/>.

10.2. Informative References

[DisProx] Cheshire, S., "Discovery Proxy for Multicast DNS-Based Service Discovery", draft-ietf-dnssd-hybrid-10 (work in progress), March 2019.

[I-D.ietf-tcpm-rack] Cheng, Y., Cardwell, N., Dukkipati, N., and P. Jha, "RACK: a time-based fast loss detection algorithm for TCP", draft-ietf-tcpm-rack-05 (work in progress), April 2019.

[LLQ] Cheshire, S. and M. Krochmal, "DNS Long-Lived Queries", draft-sekar-dns-llq-03 (work in progress), March 2019.

[obs] "Observer Pattern", <https://en.wikipedia.org/wiki/Observer_pattern>.

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

[RFC3123] Koch, P., "A DNS RR Type for Lists of Address Prefixes (APL RR)", RFC 3123, DOI 10.17487/RFC3123, June 2001, <https://www.rfc-editor.org/info/rfc3123>.

[RFC4287] Nottingham, M., Ed. and R. Sayre, Ed., "The Atom Syndication Format", RFC 4287, DOI 10.17487/RFC4287, December 2005, <https://www.rfc-editor.org/info/rfc4287>.

[RFC4953] Touch, J., "Defending TCP Against Spoofing Attacks", RFC 4953, DOI 10.17487/RFC4953, July 2007, <https://www.rfc-editor.org/info/rfc4953>.

[RFC5077] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 5077, DOI 10.17487/RFC5077, January 2008, <https://www.rfc-editor.org/info/rfc5077>.

[RFC6281] Cheshire, S., Zhu, Z., Wakikawa, R., and L. Zhang, "Understanding Apple's Back to My Mac (BTMM) Service", RFC 6281, DOI 10.17487/RFC6281, June 2011, <https://www.rfc-editor.org/info/rfc6281>.

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

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

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <https://www.rfc-editor.org/info/rfc6824>.

[RFC6886] Cheshire, S. and M. Krochmal, "NAT Port Mapping Protocol (NAT-PMP)", RFC 6886, DOI 10.17487/RFC6886, April 2013, <https://www.rfc-editor.org/info/rfc6886>.

[RFC6887] Wing, D., Ed., Cheshire, S., Boucadair, M., Penno, R., and P. Selkirk, "Port Control Protocol (PCP)", RFC 6887, DOI 10.17487/RFC6887, April 2013, <https://www.rfc-editor.org/info/rfc6887>.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <https://www.rfc-editor.org/info/rfc7413>.

[RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <https://www.rfc-editor.org/info/rfc7525>.

[RFC7719] Hoffman, P., Sullivan, A., and K. Fujiwara, "DNS Terminology", RFC 7719, DOI 10.17487/RFC7719, December 2015, <https://www.rfc-editor.org/info/rfc7719>.

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

[RFC8010] Sweet, M. and I. McDonald, "Internet Printing Protocol/1.1: Encoding and Transport", STD 92, RFC 8010, DOI 10.17487/RFC8010, January 2017, <https://www.rfc-editor.org/info/rfc8010>.

[RFC8011] Sweet, M. and I. McDonald, "Internet Printing Protocol/1.1: Model and Semantics", STD 92, RFC 8011, DOI 10.17487/RFC8011, January 2017, <https://www.rfc-editor.org/info/rfc8011>.

[RFC8310] Dickinson, S., Gillmor, D., and T. Reddy, "Usage Profiles for DNS over TLS and DNS over DTLS", RFC 8310, DOI 10.17487/RFC8310, March 2018, <https://www.rfc-editor.org/info/rfc8310>.

[SYN] Eddy, W., "Defenses Against TCP SYN Flooding Attacks", The Internet Protocol Journal, Cisco Systems, Volume 9, Number 4, December 2006.

[XEP0060] Millard, P., Saint-Andre, P., and R. Meijer, "Publish-Subscribe", XSF XEP 0060, July 2010.

Authors' Addresses

Tom Pusateri
Unaffiliated
Raleigh, NC 27608
USA

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

Stuart Cheshire
Apple Inc.
One Apple Park Way
Cupertino, CA 95014
USA

Phone: +1 (408) 996-1010
Email: cheshire@apple.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: April 26, 2019

T. Lemon
Nibbhaya Consulting
D. Migault
Ericsson
S. Cheshire
Apple Inc.

October 23, 2018

Homenet Naming and Service Discovery Architecture
draft-ietf-homenet-simple-naming-03

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

Copyright Notice

Table of Contents

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, optionally, more than one internet service provider. Home network users are assumed not to be knowledgeable 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.

This 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. Not all requirements are addressed by this architecture document, but the basic requirements are satisfied, and this document serves as a foundation upon which solutions to the remaining problems can be built.

2. Requirements

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

o Name: a forward domain under which information about local services will be published

o 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

o Resolution: a full-service caching DNS resolver that fully supports EDNS(0) and queries with the DO bit set

o 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.

2.1.1. Multiple Provisioning Domains

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].

When a host on a homenet connects to a host outside the homenet, and the homenet is multihomed, the source address that the host uses for connecting determines which upstream ISP connection is used. In principle, this is not a problem, because the Internet is a fully connected network, so any host that is on the Internet can be reached by any host on the Internet, regardless of how that host connects to the Internet.

Unfortunately in practice this is not always the case. Some ISPs provide special services to their end users that are only accessible when connected through the ISP. When such a service is discovered using that ISP’s name server, a response will be provided that will only work if the host connects using a prefix provided by that ISP. If another ISP’s prefix is used, the connection will fail.

In the case of content delivery networks (CDNs), using the name service of one ISP and then connecting through a second ISP may seem to work, but may provide very poor service.

In order to address this problem, the homenet naming architecture takes two approaches. First, for hosts that do not support provisioning domain separation, we make sure that all ISP name servers are consulted in such a way that Happy Eyeballs will tend to work. Second, for hosts that do support provisioning domain separation, we provide information to the hosts to identify provisioning domains, and we provide a mechanism that hosts can use to indicate which provisioning domain to use for a particular DNS query.

2.2. Homenet-specific considerations

A naming architecture for homenets therefore adds the following considerations:

- o All of the operations mentioned here must reliably function automatically, without any user intervention or debugging.

o Because user intervention cannot be required, naming conflicts must be resolved automatically, and, to the extent possible, transparently.

o 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.

o 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 [RFC7556] 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

Forward Mapping A mapping between a host name or service name and some information about that host or service.

Reverse Mapping A mapping between an IP address and the host that has that IP address.

Homenet Domain A domain name that is used for publishing the names of devices and services that are present on the homenet. By default, 'home.arpa.'

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 publish names for forward lookups under the reserved domain 'home.arpa.' [RFC8375] publishing names.

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.

How to publish names outside of the homenet is out of scope for this document. However, in order to address the problem of validating names published on the homenet using DNSSEC, it is necessary that the homenet have a globally valid delegation from the root. This allows hosts on the homenet to validate names published on the homenet using the DNS root trust anchor ([RFC4033] section 3.1).

It is not necessary that this delegation work for hosts off the homenet. HNRs implementing this specification do not answer queries from outside the homenet; however, when a validating resolver inside the homenet attempts to validate the chain of trust up to the root zone, the chain of trust will validate correctly, because the answer given for internally-available zones will be signed by a DS record that is present in the delegation externally.

If there is a valid delegation from the root, the homenet domain will be the name of the delegated domain. By default, there will be no delegation from the root; in this case, the homenet domainname will be 'home.arpa.'

In addition to the homenet domain, 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 [RFC1034] section 5.2.1 should be constructed. For example, if the homenet is allocating local IP addresses out of net 10 [RFC1918], 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. The reverse mapping domain for hosts on any link in the subnet will be a subdomain of the reverse zone for the subset of the ULA prefix that is being advertised on that link. 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 fc00:2001: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

There are two types of authoritative name service on the homenet. Every link on the homenet has a zone that is a subdomain of the homenet's primary domain. Authority for these zones is local to the HNR that is currently authoritative for that zone. The contents of these zones are served using DNSSD Discovery Proxy [I-D.ietf-dnssd-hybrid]. Consequently, there is no need for database replication in the case that a new HNR is elected; that HNR simply takes over the Discovery Relay function.

Name service for the homenet domain itself may be stateless or stateful. HNRs are not required to implement stateful service. If one or more HNRs on the homenet are capable of providing this service, then one of those HNRs is elected to act as the primary nameserver for the homenet domain; one or more HNRs may also act as secondary servers.

Name service for reverse mapping subdomains is only provided if one or more HNRs can provide stateful service. If no such server is present, the reverse mapping subdomains are not served. If stateful servers are present, the primary and secondary servers for these subdomains will be the same as for the homenet domain.

5.1. Reachability

Whether the homenet domain is a global domain name or not, HNRs answering queries for domains on the homenet do not respond to queries from off the homenet unless configured to do so. Exposing services on the homenet for browsing off the homenet creates many opportunities for security issues; as such, even an HNR configured to answer queries from prefixes off the homenet do not provide answers for names of devices on the homenet unless configured to do so. How reachability of names published on the homenet is managed is out of scope for this document: an HNR implementing only this document checks the source address of every query to see if it is within a prefix belonging to the homenet; if not, the HNR does not answer the query.

5.2. Link Names

Each link must have a name. These names are determined using HNCP. Each router will have zero or more wired links, each of which must be labeled. In addition, each router will have zero or more wireless links. Each of these links will be named by the frequency band the link supports, 2.4ghz or 5ghz.

The HNR is named using its manufacturer name. If, as is likely, two or more HNRs from the same manufacturer are present on a homenet, then the HNR name is made up of the manufacturer name plus as many hexadecimal digits as are required from the HNRs link layer addresses so as to disambiguate them.

When shipping multiple HNRs as a kit, manufacturers are advised to arrange that each HNR has a different number in the lowest four bits of the link-layer address. Manufacturers are also advised to print that link layer address, in full, somewhere on the outside of the HNR where it can be seen by the user. Since most HNRs will have more than one interface, the manufacturer should be consistent in choosing which link-layer address is printed on the outside and used to identify the router.

The name given to a link is the name of the HNR, plus a hyphen ('-'), plus name of the interface of that HNR that is attached to the link. In the event that this name must be displayed to the user, this should give the user enough information to figure out which link is being referenced. In the event that the HNR that is providing authoritative service for that link changes, the link name changes. This should only happen if the network topology changes.

If the appearance of a new HNR requires that the name of an existing HNR change, then the names of all the links managed by that existing HNR change to reflect the new name.

5.3. Authoritative name service for the homenet domain

All HNRs must be capable of providing authoritative name service for the homenet domain. HNRs that provide only stateless authoritative service publish the information that is required for hosts to do DNS Service Discovery over DNS, using the local resolver as a DNSSD Discovery Broker.

Some contents are required for the homenet domain, whether it is stateful or stateless.

- o Every link on the homenet has a name that is a subdomain of the homenet domain. The zone associated with the homenet domain contains a delegation for each of these subdomains.

- o In order for DNSSD service discovery to work, a default browsing domain must be published. The default browsing domain is simply the homenet domain.

- o If DNSSD SRP is supported (that is, if stateful authoritative service is present), then an SRV record must be published, along with a list of available registration zones containing exactly one entry, for the homenet domain ([I-D.sctl-service-registration] section 2).

- Also if DNSSD SRP is supported, then one or more A and/or AAAA records must be published under the name that the SRV record points to, which should be a single label subdomain of the homenet domain.

Both stateful and stateless authoritative servers provided by HNRs must support DNS Stateful Operations [I-D.ietf-dnsop-session-signal] and DNS Push [I-D.ietf-dnssd-push] for the names for which they are authoritative.

5.4. Authoritative name service for per-link subdomains of the homenet domain

Per-link subdomains of the homenet domain are served by DNSSD Discovery Proxies. Although these proxies generally do caching, no long-lived state is kept by them. DNSSD Discovery Proxies running on HNRs must support DNS Stateful Operations and DNS Push.

5.5. Authoritative name service for the ULA reverse mapping domain

The ULA reverse mapping domain itself is only published if stateful name service is available. It is represented as a single zone, which contains no delegations: every reverse mapping for an address in the ULA prefix is simply published in the ULA zone.

In order to permit registration of reverse mappings in this domain, it must contain an SRV record for the label _homenet-rrp._tcp at the top level, pointing to the primary server for the domain.

5.6. Authoritative name service for the RFC1918 reverse mapping domains

If IPv4 service is being provided on the homenet, and if stateful name service is being provided on the homenet, then either one or sixteen reverse mapping zones for the RFC1918 prefix in use must be provided. If more than one RFC1918 prefix is in use, reverse mapping zones for all such prefixes must be provided.

Like the ULA reverse mapping zone, the RFC1918 reverse mapping zones must each contain an SRV record on the label _homenet-rrp._tcp at the top level, pointing to the name of the primary server for the zone.

The RFC1918 reverse mapping zone contains the entire address space of the RFC1918 prefix that is in use on the homenet. Section 3 of RFC1918 defines three prefixes that may be used. The homenet will use all of one of these three prefixes. Of these, the 172.16.0.0 prefix is subdivided on a 12-bit boundary, and therefore must be represented as 16 separate zones. The 10.0.0.0/8 and 192.168.0.0/16 prefixes are each represented as a single zone.

The zone to be updated is therefore the 10.in-addr.arpa zone for all addresses in 10.0.0.0/8, and the 168.192.in-addr.arpa zone for all addresses in 192.168.0.0/16. For addresses in the 172.16.0.0/12 prefix, the zone to be updated is the subdomain of 172.in-addr.arpa that corresponds to bits 8-11 of the prefix: a number between 16 and 31, inclusive.

Also like the ULA zone, the RFC1918 reverse mapping zones contain no delegations: if there is a single zone, then every reverse mapping published for an address in the RFC1918 prefix in use on the homenet is published directly under this zone. If there are sixteen zones, each address is published in its respective zone. Because the zone 172.in-addr.arpa is not available to be served locally, its locally served subdomains are simply served individually with no delegation.

6. Resolution

Name resolution on the homenet must accomplish two tasks: resolving names that are published on the homenet, and resolving names that are published elsewhere. This is accomplished by providing several functional layers.

1. The set of caching nameservers provided by the ISP or ISPs through which the homenet gains access to the global internet, if any (homenets can operate standalone as well).

2. The set of stateful name servers on the homenet that are authoritative for the homenet domain as a whole, and for any reverse mapping zones that are provided by the homenet. This layer is optional, and may or may not be present. If present, it is provided by one or more HNRs on the homenet that support stateful service.

3. The set of stateless name servers on the homenet that are authoritative for the homenet domain as a whole. These are not used if one or more stateful servers are present.

4. The set of stateless DNSSD Discovery Proxies that are authoritative for each of the links in the homenet.

5. A DNS routing proxy. Hereafter we refer to this as the DNS proxy.

The reason that these are described as layers is that it's quite possible that all of the DNS services on the homenet might be provided by a single service listening on port 53; how the request is routed then depends on the question being asked. So the services described as running on HNRs are treated as functional blocks which may be connected internally, if the question being asked can be answered directly by the HNR that received it, or they may be separate name servers running on different HNRs, if the question can be answered within the homenet, or it may be that the HNR receiving the query forwards it to an ISP caching name server.

The routing works as follows. When a request is received (opcode=0, Q/R=0), the DNS proxy looks at the owner name in the question part of the message.

o If the name is a subdomain of the homenet domain, the query is local.

o If the name is a subdomain of a locally-valid ULA reverse mapping domain, the query is local.

o If the name is a subdomain of a locally valid RFC1918 reverse mapping zone, the query is local.

o If the name is a subdomain of any locally-served zone, as defined in Locally Served DNS Zones [localzones], but is not otherwise identified as local, the response is NXDOMAIN.

o Otherwise, the query is not local.

Local queries are further divided. If the query is for a link subdomain, the DNS proxy consults the table that maps per-link subdomains to the HNRs that serve them. Either the HNR that serves this link subdomain is the HNR that received the question, or not. If it is, then the DNS proxy passes the query directly to the local DNSSD Discovery Proxy. Otherwise, it forwards the query to the DNSSD Discovery Proxy on the HNR that is providing Discovery Proxy service for that link.

If the query is for the homenet subdomain, and stateful authoritative service for the homenet subdomain is present on the homenet, then either the HNR receiving the query provides stateful authoritative service, or not. If it does, then the query is passed directly to the local authoritative server. If not, then the HNR looks in the table of authoritative servers generated by HNCP and forwards the request to one of these servers. Queries for the reverse mapping zones are handled the same way.

Otherwise, the query is examined to see if it contains an EDNS(0) Provisioning Domain option. If not, it round-robined across the resolvers provided by each ISP in such a way that each ISP is tried in succession, and the same ISP is not asked the same question repeatedly. If the query does contain the EDNS(0) Provisioning Domain option, then that option is used to select which ISP's resolvers are used for the round robin.

6.1. Round Robining

There are several cases above where there may be a choice of servers to which to forward queries. It's assumed that when the query can be satisfied by the HNR that received it, round robining is not required. If there is a specific HNR that is responsible for a particular link, then round robining is likewise not required. However, if the query is for a stateful authoritative server, and the HNR that received it does not provide this service, and there is more than one HNR on the homenet that does provide the service, the HNR that received the query round robins it across the available set of HNRs that could answer it.

Similarly, if the query is to be sent to an ISP's resolver, and the ISP has provided more than one resolver, round robining is done across the set of resolvers provided by that ISP. If the query is to be attempted at every ISP, then that is accomplished by round-robining in such a way that each ISP is tried in succession, rather than all the resolvers at one ISP, and then all the resolvers at the next ISP, and so on.

6.2. Retransmission

For queries that can't be resolved locally by the HNR that received them, retransmission as described in [RFC1035] is performed.

6.3. DNS Stateful Operations and DNS Push

DNS proxies on HNRs are required to support DNS Stateful Operations and DNS Push. When a DNS Push operation is requested on a name that can be satisfied by the HNR that received it, it is handled locally. When such an operation is requested on a name that is local to the homenet, but can't be satisfied by the HNR that received it, a DNS Stateful operation is started with the HNR that is responsible for it.

6.4. Multicast DNS

In addition to consulting the local resolver, hosts on the homenet may attempt to discover services directly using Multicast DNS. HNRs may filter out incoming Multicast DNS queries, forcing the client to do service discovery using the DNS protocol. If such filtering is not done, the client will be able to discover services on the link to which it is attached, but will not be able to discover services elsewhere.

It is believed that all currently-available hosts support DNSSD using the DNS protocol. Support for mDNS on the local link is therefore not required. However, if an mDNS query returns the same answer as the DNS protocol query, this is not expected to be a problem.

6.5. Host behavior

Hosts that support the RA Provisioning Domain option direct queries to the name server(s) of the provisioning domain they will use for communication using the EDNS(0) provisioning domain option. In practice this means that a host that supports PvDs will keep a set of provisioning information for each prefix that it received from the router, and will either choose a prefix to use based on its own criteria, or will attempt to connect using every PvD at once or in sequence. Answers to queries sent for a particular provisioning domain will only be used with source addresses for prefixes that are in that provisioning domain.

7. Publication

Names are published either using Multicast DNS Service Discovery [RFC6762] or DNSSD Service Registration Protocol ([I-D.sctl-service-registration]). Reverse mappings are published using Homenet Reverse Mapping Update Protocol Section 7.2.

7.1. DNSSD Service Registration Protocol

HNRs that provide stateful authoritative service also publish information acquired using DNSSD Service Registration Protocol [I-D.sctl-service-registration]. DNSSD SRP does not explicitly support population of the reverse zone; hosts that wish to provide reverse mapping information must first establish their hostname using DNSSD SRP; once established, the key used to authenticate the DNSSD SRP update is also used to update the reverse name.

Support for SRP provides several advantages over DNSSD Discovery Proxy. First, DNSSD SRP provides a secure way of claiming service names. Second, a claimed name is valid for the entire network covered by the SRP service, not just an individual link, as is the case with mDNS. Third, SRP does not use multicast, and is therefore more reliable on links with constrained multicast support [I-D.ietf-mboned-ieee802-mcast-problems].

Support for the DNSSD SRP service is not sufficient to achieve full deployment of DNSSD SRP: it is also necessary that services advertise using DNSSD SRP. Requiring such support is out of scope for this document; our goal is simply to specify a way in which DNSSD SRP can be supported on homenets, so that that as adoption of SRP increases on devices providing service, it can actually be used.

7.2. Homenet Reverse Mapping Update Protocol

This is an extension to the DNSSD Service Registration protocol. The purpose is to allow for updates of reverse mappings. Hosts wishing to publish reverse mappings first publish their hostname using DNSSD SRP. When this process has successfully completed, the host can add reverse mappings to the ULA reverse mapping domain and to the RFC1918 reverse mapping domain, if present.

7.2.1. Adding ULA reverse mappings

The host first determines the ULA prefix. If there is more than one ULA prefix active, the ULA prefix with the longest preferred lifetime is used. A ULA prefix can be identified because it matches the prefix fc00::/7 ([RFC4193] section 3.1). The actual prefix is then the first 48 bits of the advertised prefix or the IP address in that prefix.

Because the ULA reverse mapping zone contains no delegations, all updates go to that one zone. To determine where to send the updates, the host first queries the SRV record under the label _homenet-rrp._tcp at the top of the ULA reverse mapping zone. It then uses the name contained in the SRV record to look up A and/or AAAA records to which to send the update.

The update is then signed using SIG(0) with the key that was used for the DNSSD SRP registration. The update is then sent using DNS Update [RFC2136] to one of the IP addresses received during the A/AAAA resolution step. The update is sent using TCP; if a TCP connection to one of the addresses fails, each subsequent address is tried in succession; if none of the addresses is reachable, the update fails, and may be retried after a reasonable period (on the order of an hour) has elapsed.

7.2.2. Adding RFC1918 reverse mappings

RFC1918 reverse mapping updates use the same mechanism as ULA reverse mapping updates. The host must first determine which zone to update, as described earlier in Section 5.6. Once the zone has been determined, the reverse mapping is updated as described in Section 7.2.1.

8. Host Configuration

Each HNR provides a Homenet DNS Proxy. When an HNR provides the DNS resolver IP address to hosts on the link using RA, DHCPv4 or DHCPv6, it provides its own address. The IPv4 address that it provides is a valid IPv4 address on the link to which the host is attached. The IPv6 address it provides is an address in the homenet's ULA prefix that is valid on the link to which the host is attached.

When sending router advertisements, the homenet includes the PvD ID RA option [I-D.ietf-intarea-provisioning-domains] in each RA. Because the PvD ID RA option can only be sent once per RA message, if the homenet is connected to more than one ISP, the prefixes for each ISP must be advertised in different RA options. In this case, the prefix for the ULA should also be sent in a separate RA.

If the configuration received from the ISP includes a Domain Name (DHCPv4) or Domain Search List (DHCPv4 or DHCPv6) option, the domain name provided is used to identify the PvD. In the case of Domain Search List options, if there is more than one, the first one is used. For the ULA prefix, the homenet domain is used to identify the PvD.

In order to facilitate DNSSD bootstrapping, any DHCPv4, DHCPv6 or RA Domain Search List options contain only a single domain name, the homenet domain. This allows hosts to quickly bootstrap DNS Service Discovery using the local domain name, as descried in [RFC6763] section 11.

9. Globally Unique Names

Homenets are not required to have globally unique names. Homenets operating according to this specification do not publish names in such a way that they can be resolved by hosts that aren't on the homenet. However, such names are useful for DNSSEC validation.

There are three ways that homenets can get global names:

1. They can be manually configured by the user. How this is done is out of scope for this document.

2. They can publish a delegation with the ISP, using a Homenet Delegation Registration Protocol Section 11.

3. They can publish a delegation with some other provider, using Homenet Delegation Registration Protocol Section 11. How this is configured is out of scope for this document.

Homenets are also not required to support global delegations for reverse mapping of global IPv4 and IPv6 addresses. How this would be done is out of scope for this document.

10. DNSSEC Validation

DNSSEC validation for 'home.arpa' requires installing a per-homenet trust anchor on the host, and is therefore not practical. Validation for locally-served reverse zones for the ULA and RFC1918 addresses would likewise require a trust anchor to be installed on the host, and likewise are not practical.

If DNSSEC validation is to be done for the homenet, the homenet must acquire a global name, and must be provided with a secure delegation. Secure delegations must also be provided from the homenet domain to each of the per-link subdomains.

Each HNR on a homenet generates its own private/public key pair that can serve as a trust anchor. These keys are shared using HNCP [RFC7788]. HNRs MUST NOT use pre-installed keys: each HNR MUST generate its own key. The HNR responsible for authoritative Discovery Proxy service on a particular link signs the zone for that link; delegations from the homenet domain zone to each per-link subdomain zone include a DS record signed by the ZSK of the homenet zone.

10.1. How trust is established

Every HNR has its own public/private key pair. A DS record for each such public key is published in the delegation for the homenet domain. If stateless authoritative service for the homenet zone is being provided, then each HNR signs its own homenet zone. The signed zone should be very stable, although the delegations may change when the network topology changes. The HNR can therefore sign the zone using its private key whenever it changes. Each HNR will have a copy of the zone signed with a different key, but since all of the ZSKs are present in the DS RRset at the delegation point, validation will succeed.

If stateful authoritative service is being provided, the HNR that is acting as primary signs the zone, and all the secondaries serve copies acquired using zone transfers. If the HNR that is primary goes away, then a secondary becomes primary and signs the zone before beginning to provide service. Again, since all of the HNR's public keys exist in the DS RRset at the delegation, the zone can be validated.

11. Homenet Delegation Registration Protocol

Homenet Delegation Registration Protocol (HDeRP) operates similarly to DNSSD Service Registration Protocol. When a homenet is not connected to an ISP that supports HDeRP, and then an ISP connection becomes available, the HNR that is connected to the ISP determines whether HDeRP is available. This is done by first determining the ISP domain.

If the connection to the ISP is IPv4-only, this will be either the DHCPv4 Domain Name option or, if not present, the only domain name in the DHCPv4 Domain Name Search List option. If the Domain Name Search List option contains more than one name, HDeRP is not supported by the ISP.

If the connection to the ISP is dual-stack or IPv6-only, then the DHCPv6 Domain Search List option obtained through DHCPv6 Prefix Delegation is used. If it is not present, or if it contains more than one domain name, HDeRP is not supported by the ISP.

Once the ISP domain has been discovered, the HNR looks for an SRV record owned by the name _homenet-derp._tcp under the ISP domain. If this is not present, HDeRP is not supported. If the SRV record is present, then the HNR looks for A and AAAA records on the hostname provided in the HNR. If present, these are used when attempting the update.

The HNR then constructs a DNS update. The DNS update creates a delegation for the zone home.arpa, with a DS record for each HNR on the homenet, containing that HNR's public key. The HNR doing the update lists its key as the first key in the DS RRset. The update is signed using SIG(0) with the private key of the HNR that is constructing it. As with DNSSD SRP, the update includes an Update Lease EDNS(0) option, specifying a key lifetime of a week.

The HNR then attempts to connect to the hostname provided in the SRV record, in a round-robin fashion across the set of IP addresses discovered during the A/AAAA lookup phase. When it has successfully connected, it sends the DNS update.

The HDeRP server validates the update by checking the SIG(0) signature of the update against the first key in the DS RRset. If the update is successfully validated, then the server generates a domain name and sends a reply back to the HNR on the same TCP connection, including the NOERROR (0) RCODE, and including in the query section the actual domain name that was generated.

This domain name then becomes the homenet name. Subsequent updates use the homenet name rather than 'home.arpa'. It is not necessary that the same HNR do the update; if a different HNR does the update, it lists its public key first in the DS RRset, and signs the update using its private key.

The HDeRP is responsible for removing the delegation if it is not refreshed for the length of its lifetime. HNRs should attempt to refresh the delegation when half the lifetime has experienced, then again at 5/8ths, and again at 7/8ths of the lifetime. If the ISP becomes unavailable, and a different ISP becomes available that supports HDeRP, the homenet should migrate to the new ISP.

12. Using the Local Namespace While Away From Home

This document does not specify 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. Expected Host Behavior

It is expected that hosts will fall into one of two categories: hosts that are able to discover DNS-SD browsing domains, and hosts that are not. Hosts that can discover DNS-SD browsing domains can be expected to successfully use service discovery across the entire homenet. Hosts that do not will only be able to discover services on the particular local subnet of the homenet to which they happen to be attached at any given time.

This is not considered to be a problem, since it is understood by the authors that the vast majority of hosts that are capable of doing mDNS discovery are also capable of doing DNS-SD discovery as described in [RFC6763].

14. 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.

15. 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.

16. 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.

17. IANA considerations

17.1. Homenet Reverse Registration Protocol

IANA is requested to add a new entry to the Service Names and Port Numbers registry for homenet-rrp with a transport type of tcp. No port number is to be assigned. The reference should be to this document, and the Assignee and Contact information should reference the authors of this document. The Description should be as follows:

Availability of Homenet Reverse Registration Protocol service for a given domain is advertised using an SRV record with an owner name of "_homenet-rrp._tcp.<domain>." in that domain, which gives the target host and port where Homenet Reverse Registration service is provided for the named domain.

17.2. Homenet Delegation Registration Protocol

IANA is requested to add a new entry to the Service Names and Port Numbers registry for homenet-derp with a transport type of tcp. No port number is to be assigned. The reference should be to this document, and the Assignee and Contact information should reference the authors of this document. The Description should be as follows:

Availability of Homenet Delegation Registration Protocol service for a given domain is advertised using an SRV record with an owner name of "_homenet-derp._tcp.<domain>." in that domain, which gives the target host and port where Homenet Delegation Registration service is provided for the named domain.

17.3. Unique Local Address Reserved Documentation Prefix

IANA is requested to add an entry to the IPv6 Special-Purpose Address Registry for the prefix fc00:2001:db8::/48. The Name shall be "Unique Local Address Documentation Prefix." The reference RFC will be this document, once published. The date will be the date the entry was added. All other fields will be the same as for the Documentation prefix, 2001:db8::/32.

18. References

18.1. Normative References

[I-D.ietf-dnsop-session-signal]
Bellis, R., Cheshire, S., Dickinson, J., Dickinson, S., Lemon, T., and T. Pusateri, "DNS Stateful Operations", draft-ietf-dnsop-session-signal-16 (work in progress), September 2018.

[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-15 (work in progress), September 2018.

[I-D.ietf-intarea-provisioning-domains]
Pfister, P., Vyncke, E., Pauly, T., Schinazi, D., and W. Shao, "Discovering Provisioning Domain Names and Data", draft-ietf-intarea-provisioning-domains-03 (work in progress), October 2018.

[I-D.sctl-service-registration]
Cheshire, S. and T. Lemon, "Service Registration Protocol for DNS-Based Service Discovery", draft-sctl-service-registration-02 (work in progress), July 2018.

[localzones]
Internet Assigned Numbers Authority, "Locally-Served DNS Zones", n.d., <https://www.iana.org/assignments/locally-served-dns-zones/locally-served-dns-zones.xhtml>.

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

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

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

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

[RFC4033] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "DNS Security Introduction and Requirements", RFC 4033, DOI 10.17487/RFC4033, March 2005, <https://www.rfc-editor.org/info/rfc4033>.

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

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

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

[RFC7336] Peterson, L., Davie, B., and R. van Brandenburg, Ed., "Framework for Content Distribution Network Interconnection (CDNI)", RFC 7336, DOI 10.17487/RFC7336, August 2014, <https://www.rfc-editor.org/info/rfc7336>.

[RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <https://www.rfc-editor.org/info/rfc7556>.

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

[RFC8375] Pfister, P. and T. Lemon, "Special-Use Domain 'home.arpa.'", RFC 8375, DOI 10.17487/RFC8375, May 2018, <https://www.rfc-editor.org/info/rfc8375>.

18.2. Informative References

[I-D.ietf-mboned-ieee802-mcast-problems]
Perkins, C., McBride, M., Stanley, D., Kumari, W., and J. Zuniga, "Multicast Considerations over IEEE 802 Wireless Media", draft-ietf-mboned-ieee802-mcast-problems-02 (work in progress), August 2018.

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

Internet Engineering Task Force S. Cheshire
Internet-Draft Apple Inc.
Intended status: Informational T. Lemon
Expires: January 3, 2018 Nominum, Inc.
July 2, 2017

Service Discovery Broker
draft-sctl-discovery-broker-00

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.

Status of This Memo

Copyright Notice

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.

```
client -> proxy

client -> broker -> proxy

client -> broker -> broker -> proxy
```

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:

```
------------                  ---------------     -------------
| Ethernet |        -->       | Aggregating | --> | Discovery |
|  Client  |                  |   Broker    |     |   Proxy   |
------------                  ---------------     -------------

------------     -------------     ---------------     -------------
|  Wi-Fi   | --> | Filtering | --> | Aggregating | --> | Discovery |
|  Client  |     |  Broker   |     |   Broker    |     |   Proxy   |
------------     -------------     ---------------     -------------
```

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

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

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

[RFC6760] Cheshire, S. and M. Krochmal, "Requirements for a Protocol to Replace the AppleTalk Name Binding Protocol (NBP)", RFC 6760, DOI 10.17487/RFC6760, February 2013, <http://www.rfc-editor.org/info/rfc6760>.

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

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

[Roadmap] Cheshire, S., "Service Discovery Road Map", draft-cheshire-dnssd-roadmap-00 (work in progress), July 2017.

[DisProx] Cheshire, S., "Discovery Proxy for Multicast DNS-Based Service Discovery", draft-ietf-dnssd-hybrid-06 (work in progress), March 2017.

[Push] Pusateri, T. and S. Cheshire, "DNS Push Notifications", draft-ietf-dnssd-push-12 (work in progress), July 2017.

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

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

Network Working Group S. Cheshire
Internet-Draft Apple Inc.
Intended status: Standards Track T. Lemon
Expires: September 19, 2018 Nibbhaya Consulting
March 18, 2018

Multicast DNS Discovery Relay
draft-sctl-dnssd-mdns-relay-04

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

Copyright Notice

Table of Contents

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 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.

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 Link Identifier TLV can only be used as an additional TLV.

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:

```
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.

Opcode	Status	Name
TBD-R	Standard	mDNS Link Request
TBD-D	Standard	mDNS Discontinue
TBD-L	Standard	Link Identifier
TBD-M	Standard	mDNS Messsage
TBD-2	Standard	Layer Two Source Address
TBD-A	Standard	IP Source

DSO Type Codes to be allocated

12. Acknowledgments

13. References

13.1. Normative References

[I-D.ietf-dnsop-session-signal]
Bellis, R., Cheshire, S., Dickinson, J., Dickinson, S., Lemon, T., and T. Pusateri, "DNS Stateful Operations", draft-ietf-dnsop-session-signal-06 (work in progress), March 2018.

[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]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-26 (work in progress), March 2018.

[I-D.sctl-discovery-broker]
Cheshire, S. and T. Lemon, "Service Discovery Broker", draft-sctl-discovery-broker-00 (work in progress), July 2017.

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

[RFC1323] Jacobson, V., Braden, R., and D. Borman, "TCP Extensions for High Performance", RFC 1323, DOI 10.17487/RFC1323, May 1992, <https://www.rfc-editor.org/info/rfc1323>.

[RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed., and A. Bierman, Ed., "Network Configuration Protocol (NETCONF)", RFC 6241, DOI 10.17487/RFC6241, June 2011, <https://www.rfc-editor.org/info/rfc6241>.

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

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

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

13.2. Informative References

[AdFam] "IANA Address Family Numbers Registry", <https://www.iana.org/assignments/address-family-numbers/>.

[NOTSENT] "TCP_NOTSENT_LOWAT socket option", July 2013, <https://lwn.net/Articles/560082/>.

[PRIO] "Prioritization Only Works When There’s Pending Data to Prioritize", January 2014, <https://insouciant.org/tech/prioritization-only-works-when-theres-pending-data-to-prioritize/>.

[TR-069] Broadband Forum, "CPE WAN Management Protocol", November 2013, <https://www.broadband-forum.org/technical/download/TR-069_Amendment-5.pdf>.

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

Internet Engineering Task Force S. Cheshire
Internet-Draft Apple Inc.
Intended status: Informational T. Lemon
Expires: January 15, 2019 Nibbhaya Consulting
July 14, 2018

Service Registration Protocol for DNS-Based Service Discovery
draft-sctl-service-registration-02

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.

Status of This Memo

Copyright Notice

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.

- o Service Discovery records are one or more PTR RRs, mapping from the generic service type (or subtype) to the specific Service Instance Name.

- o 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.

- o 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:

- o It implements first-come first-served name allocation, protected using SIG(0).
- o It enforces policy about what updates are allowed.
- o It optionally performs rewriting of "services.arpa" to some other domain.
- o It optionally performs automatic population of the address-to-name reverse mapping domains.
- o A DNS-SD Service Registration server is not required to implement general DNS Update prerequsite processing.
- o 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

- o exactly one RRset update,
- o which is for a PTR RR,
- o which points to a Service Instance Name
- o 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

- o exactly one "Delete all RRsets from a name" update,
- o exactly one SRV RRset update,
- o one or more TXT RRset updates,
- o 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

- o exactly one "Delete all RRsets from a name" update,
- o A or AAAA RR update(s)
- o 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,
- o 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 is not 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 rejectration 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 DNSSD 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

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

[I-D.sekar-dns-ul] Sekar, K., "Dynamic DNS Update Leases", draft-sekar-dns-ul-01 (work in progress), August 2006.

6.2. Informative References

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

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

[RFC2131] Droms, R., "Dynamic Host Configuration Protocol", RFC 2131, DOI 10.17487/RFC2131, March 1997, <https://www.rfc-editor.org/info/rfc2131>.

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

[RFC2181] Elz, R. and R. Bush, "Clarifications to the DNS Specification", RFC 2181, DOI 10.17487/RFC2181, July 1997, <https://www.rfc-editor.org/info/rfc2181>.

[RFC2931] Eastlake 3rd, D., "DNS Request and Transaction Signatures (SIG(0)s)", RFC 2931, DOI 10.17487/RFC2931, September 2000, <https://www.rfc-editor.org/info/rfc2931>.

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

[RFC3152] Bush, R., "Delegation of IP6.ARPA", BCP 49, RFC 3152, DOI 10.17487/RFC3152, August 2001, <https://www.rfc-editor.org/info/rfc3152>.

[RFC6760] Cheshire, S. and M. Krochmal, "Requirements for a Protocol to Replace the AppleTalk Name Binding Protocol (NBP)", RFC 6760, DOI 10.17487/RFC6760, February 2013, <https://www.rfc-editor.org/info/rfc6760>.

[RFC6761] Cheshire, S. and M. Krochmal, "Special-Use Domain Names", RFC 6761, DOI 10.17487/RFC6761, February 2013, <https://www.rfc-editor.org/info/rfc6761>.

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

[RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <https://www.rfc-editor.org/info/rfc7228>.

[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. , ISBN 0-596-10100-7, December 2005.

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