Internet Engineering Task Force (IETF)
Request for Comments: 6763
Category: Standards Track
ISSN: 2070-1721

S. Cheshire
M. Krochmal
Apple Inc.

February 2013

DNS-Based Service Discovery

Abstract

This document specifies how DNS resource records are named and structured to facilitate service discovery. Given a type of service that a client is looking for, and a domain in which the client is looking for that service, this mechanism allows clients to discover a list of named instances of that desired service, using standard DNS queries. This mechanism is referred to as DNS-based Service Discovery, or DNS-SD.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6763.

Table of Contents

1. Introduction

This document specifies how DNS resource records are named and structured to facilitate service discovery. Given a type of service that a client is looking for, and a domain in which the client is looking for that service, this mechanism allows clients to discover a list of named instances of that desired service, using standard DNS queries. This mechanism is referred to as DNS-based Service Discovery, or DNS-SD.

This document proposes no change to the structure of DNS messages, and no new operation codes, response codes, resource record types, or any other new DNS protocol values.

This document specifies that a particular service instance can be described using a DNS SRV [RFC2782] and DNS TXT [RFC1035] record. The SRV record has a name of the form "<Instance>.<Service>.<Domain>" and gives the target host and port where the service instance can be reached. The DNS TXT record of the same name gives additional information about this instance, in a structured form using key/value pairs, described in Section 6. A client discovers the list of available instances of a given service type using a query for a DNS PTR [RFC1035] record with a name of the form "<Service>.<Domain>", which returns a set of zero or more names, which are the names of the aforementioned DNS SRV/TXT record pairs.

This specification is compatible with both Multicast DNS [RFC6762] and with today's existing Unicast DNS server and client software.

When used with Multicast DNS, DNS-SD can provide zero-configuration operation -- just connect a DNS-SD/mDNS device, and its services are advertised on the local link with no further user interaction [ZC].

When used with conventional Unicast DNS, some configuration will usually be required -- such as configuring the device with the DNS domain(s) in which it should advertise its services, and configuring it with the DNS Update [RFC2136] [RFC3007] keys to give it permission to do so. In rare cases, such as a secure corporate network behind a

firewall where no DNS Update keys are required, zero-configuration operation may be achieved by simply having the device register its services in a default registration domain learned from the network (see Section 11, "Discovery of Browsing and Registration Domains"), but this is the exception and usually security credentials will be required to perform DNS updates.

Note that when using DNS-SD with Unicast DNS, the Unicast DNS-SD service does NOT have to be provided by the same DNS server hardware that is currently providing an organization's conventional host name lookup service. While many people think of "DNS" exclusively in the context of mapping host names to IP addresses, in fact, "the DNS is a general (if somewhat limited) hierarchical database, and can store almost any kind of data, for almost any purpose" [RFC2181]. By delegating the "_tcp" and "_udp" subdomains, all the workload related to DNS-SD can be offloaded to a different machine. This flexibility, to handle DNS-SD on the main DNS server or not, at the network administrator's discretion, is one of the benefits of using DNS.

Even when the DNS-SD functions are delegated to a different machine, the benefits of using DNS remain: it is mature technology, well understood, with multiple independent implementations from different vendors, a wide selection of books published on the subject, and an established workforce experienced in its operation. In contrast, adopting some other service discovery technology would require every site in the world to install, learn, configure, operate, and maintain some entirely new and unfamiliar server software. Faced with these obstacles, it seems unlikely that any other service discovery technology could hope to compete with the ubiquitous deployment that DNS already enjoys. For further discussion, see Appendix A, "Rationale for Using DNS as a Basis for Service Discovery".

This document is written for two audiences: for developers creating application software that offers or accesses services on the network, and for developers creating DNS-SD libraries to implement the advertising and discovery mechanisms. For both audiences, understanding the entire document is helpful. For developers creating application software, this document provides guidance on choosing instance names, service names, and other aspects that play a role in creating a good overall user experience. However, also understanding the underlying DNS mechanisms used to provide the service discovery facilities helps application developers understand the capabilities and limitations of those underlying mechanisms (e.g., name length limits). For library developers writing software to construct the DNS records (to advertise a service) and generate the DNS queries (to discover and use a service), understanding the ultimate user-experience goals helps them provide APIs that can meet those goals.

2. Conventions and Terminology Used in This Document

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

3. Design Goals

Of the many properties a good service discovery protocol needs to have, three of particular importance are:

(i) The ability to query for services of a certain type in a certain logical domain, and receive in response a list of named instances (network browsing or "Service Instance Enumeration").

(ii) Given a particular named instance, the ability to efficiently resolve that instance name to the required information a client needs to actually use the service, i.e., IP address and port number, at the very least (Service Instance Resolution).

(iii) Instance names should be relatively persistent. If a user selects their default printer from a list of available choices today, then tomorrow they should still be able to print on that printer -- even if the IP address and/or port number where the service resides have changed -- without the user (or their software) having to repeat the step (i) (the initial network browsing) a second time.

In addition, if it is to become successful, a service discovery protocol should be so simple to implement that virtually any device capable of implementing IP should not have any trouble implementing the service discovery software as well.

These goals are discussed in more detail in the remainder of this document. A more thorough treatment of service discovery requirements may be found in "Requirements for a Protocol to Replace the AppleTalk Name Binding Protocol (NBP)" [RFC6760]. That document draws upon examples from two decades of operational experience with AppleTalk to develop a list of universal requirements that are broadly applicable to any potential service discovery protocol.

4. Service Instance Enumeration (Browsing)

Traditional DNS SRV records [RFC2782] are useful for locating instances of a particular type of service when all the instances are effectively indistinguishable and provide the same service to the client.

For example, SRV records with the (hypothetical) name "_http._tcp.example.com." would allow a client to discover servers implementing the "_http._tcp" service (i.e., web servers) for the "example.com." domain. The unstated assumption is that all these servers offer an identical set of web pages, and it doesn't matter to the client which of the servers it uses, as long as it selects one at random according to the weight and priority rules laid out in the DNS SRV specification [RFC2782].

Instances of other kinds of service are less easily interchangeable. If a word processing application were to look up the (hypothetical) SRV record "_ipp._tcp.example.com." to find the list of Internet Printing Protocol (IPP) [RFC2910] printers at Example Co., then picking one at random and printing on it would probably not be what the user wanted.

The remainder of this section describes how SRV records may be used in a slightly different way, to allow a user to discover the names of all available instances of a given type of service, and then select, from that list, the particular instance they desire.

4.1. Structured Service Instance Names

This document borrows the logical service-naming syntax and semantics from DNS SRV records, but adds one level of indirection. Instead of requesting records of type "SRV" with name "_ipp._tcp.example.com.", the client requests records of type "PTR" (pointer from one name to another in the DNS namespace) [RFC1035].

In effect, if one thinks of the domain name "_ipp._tcp.example.com." as being analogous to an absolute path to a directory in a file system, then DNS-SD's PTR lookup is akin to performing a listing of that directory to find all the entries it contains. (Remember that domain names are expressed in reverse order compared to path names -- an absolute path name starts with the root on the left and is read from left to right, whereas a fully qualified domain name starts with the root on the right and is read from right to left. If the fully qualified domain name "_ipp._tcp.example.com." were expressed as a file system path name, it would be "/com/example/_tcp/_ipp".)

The result of this PTR lookup for the name "<Service>.<Domain>" is a set of zero or more PTR records giving Service Instance Names of the form:

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

For explanation of why the components are in this order, see Appendix B, "Ordering of Service Instance Name Components".

4.1.1. Instance Names

The <Instance> portion of the Service Instance Name is a user-friendly name consisting of arbitrary Net-Unicode text [RFC5198]. It MUST NOT contain ASCII control characters (byte values 0x00-0x1F and 0x7F) [RFC20] but otherwise is allowed to contain any characters, without restriction, including spaces, uppercase, lowercase, punctuation -- including dots -- accented characters, non-Roman text, and anything else that may be represented using Net-Unicode. For discussion of why the <Instance> name should be a user-visible, user-friendly name rather than an invisible machine-generated opaque identifier, see Appendix C, "What You See Is What You Get".

The <Instance> portion of the name of a service being offered on the network SHOULD be configurable by the user setting up the service, so that he or she may give it an informative name. However, the device or service SHOULD NOT require the user to configure a name before it can be used. A sensible choice of default name can in many cases allow the device or service to be accessed without any manual configuration at all. The default name should be short and descriptive, and SHOULD NOT include the device's Media Access Control (MAC) address, serial number, or any similar incomprehensible hexadecimal string in an attempt to make the name globally unique. For discussion of why <Instance> names don't need to be (and SHOULD NOT be) made unique at the factory, see Appendix D, "Choice of Factory-Default Names".

This <Instance> portion of the Service Instance Name is stored directly in the DNS as a single DNS label of canonical precomposed UTF-8 [RFC3629] "Net-Unicode" (Unicode Normalization Form C) [RFC5198] text. For further discussion of text encodings, see Appendix E, "Name Encodings in the Domain Name System".

DNS labels are currently limited to 63 octets in length. UTF-8 encoding can require up to four octets per Unicode character, which means that in the worst case, the <Instance> portion of a name could be limited to fifteen Unicode characters. However, the Unicode

characters with longer octet lengths under UTF-8 encoding tend to be the more rarely used ones, and tend to be the ones that convey greater meaning per character.

Note that any character in the commonly used 16-bit Unicode Basic Multilingual Plane [Unicode6] can be encoded with no more than three octets of UTF-8 encoding. This means that an instance name can contain up to 21 Kanji characters, which is a sufficiently expressive name for most purposes.

4.1.2. Service Names

The <Service> portion of the Service Instance Name consists of a pair of DNS labels, following the convention already established for SRV records [RFC2782]. The first label of the pair is an underscore character followed by the Service Name [RFC6335]. The Service Name identifies what the service does and what application protocol it uses to do it. The second label is either "_tcp" (for application protocols that run over TCP) or "_udp" (for all others). For more details, see Section 7, "Service Names".

4.1.3. Domain Names

The <Domain> portion of the Service Instance Name specifies the DNS subdomain within which those names are registered. It may be "local.", meaning "link-local Multicast DNS" [RFC6762], or it may be a conventional Unicast DNS domain name, such as "ietf.org.", "cs.stanford.edu.", or "eng.us.ibm.com." Because Service Instance Names are not host names, they are not constrained by the usual rules for host names [RFC1033] [RFC1034] [RFC1035], and rich-text service subdomains are allowed and encouraged, for example:

```
Building 2, 1st Floor  .  example  .  com  .
Building 2, 2nd Floor  .  example  .  com  .
Building 2, 3rd Floor  .  example  .  com  .
Building 2, 4th Floor  .  example  .  com  .
```

In addition, because Service Instance Names are not constrained by the limitations of host names, this document recommends that they be stored in the DNS, and communicated over the wire, encoded as straightforward canonical precomposed UTF-8 [RFC3629] "Net-Unicode" (Unicode Normalization Form C) [RFC5198] text. In cases where the DNS server returns a negative response for the name in question, client software MAY choose to retry the query using the "Punycode" algorithm [RFC3492] to convert the UTF-8 name to an IDNA "A-label" [RFC5890], beginning with the top-level label, then issuing the query

repeatedly, with successively more labels translated to IDNA A-labels each time, and giving up if it has converted all labels to IDNA A-labels and the query still fails.

4.2. User Interface Presentation

The names resulting from the Service Instance Enumeration PTR lookup are presented to the user in a list for the user to select one (or more). Typically, only the first label is shown (the user-friendly <Instance> portion of the name).

In the common case the <Service> and <Domain> are already known to the client software, these having been provided implicitly by the user in the first place, by the act of indicating the service being sought, and the domain in which to look for it. Note that the software handling the response should be careful not to make invalid assumptions though, since it *is* possible, though rare, for a service enumeration in one domain to return the names of services in a different domain. Similarly, when using subtypes (see Section 7.1, "Selective Instance Enumeration") the <Service> of the discovered instance may not be exactly the same as the <Service> that was requested.

For further discussion of Service Instance Enumeration (browsing) user-interface considerations, see Appendix F, "'Continuous Live Update' Browsing Model".

Once the user has selected the desired named instance, the Service Instance Name may then be used immediately, or saved away in some persistent user-preference data structure for future use, depending on what is appropriate for the application in question.

4.3. Internal Handling of Names

If client software takes the <Instance>, <Service>, and <Domain> portions of a Service Instance Name and internally concatenates them together into a single string, then because the <Instance> portion is allowed to contain any characters, including dots, appropriate precautions MUST be taken to ensure that DNS label boundaries are properly preserved. Client software can do this in a variety of ways, such as character escaping.

This document RECOMMENDS that if concatenating the three portions of a Service Instance Name, any dots in the <Instance> portion be escaped following the customary DNS convention for text files: by preceding literal dots with a backslash (so "." becomes "\."). Likewise, any backslashes in the <Instance> portion should also be escaped by preceding them with a backslash (so "\" becomes "\\").

Having done this, the three components of the name may be safely concatenated. The backslash-escaping allows literal dots in the name (escaped) to be distinguished from label-separator dots (not escaped), and the resulting concatenated string may be safely passed to standard DNS APIs like res_query(), which will interpret the backslash-escaped string as intended.

5. Service Instance Resolution

When a client needs to contact a particular service, identified by a Service Instance Name, previously discovered via Service Instance Enumeration (browsing), it queries for the SRV and TXT records of that name. The SRV record for a service gives the port number and target host name where the service may be found. The TXT record gives additional information about the service, as described in Section 6, "Data Syntax for DNS-SD TXT Records".

SRV records are extremely useful because they remove the need for preassigned port numbers. There are only 65535 TCP port numbers available. These port numbers are traditionally allocated one per application protocol [RFC6335]. Some protocols like the X Window System have a block of 64 TCP ports allocated (6000-6063). Using a different TCP port for each different instance of a given service on a given machine is entirely sensible, but allocating each application its own large static range, as was done for the X Window System, is not a practical way to do that. On any given host, most TCP ports are reserved for services that will never run on that particular host in its lifetime. This is very poor utilization of the limited port space. Using SRV records allows each host to allocate its available port numbers dynamically to those services actually running on that host that need them, and then advertise the allocated port numbers via SRV records. Allocating the available listening port numbers locally on a per-host basis as needed allows much better utilization of the available port space than today’s centralized global allocation.

In the event that more than one SRV is returned, clients MUST correctly interpret the priority and weight fields -- i.e., lower-numbered priority servers should be used in preference to higher-numbered priority servers, and servers with equal priority should be selected randomly in proportion to their relative weights. However, in the overwhelmingly common case, a single advertised DNS-SD service instance is described by exactly one SRV record, and in this common case the priority and weight fields of the SRV record SHOULD both be set to zero.

6. Data Syntax for DNS-SD TXT Records

Some services discovered via Service Instance Enumeration may need more than just an IP address and port number to completely identify the service instance. For example, printing via the old Unix LPR (port 515) protocol [RFC1179] often specifies a queue name [BJP]. This queue name is typically short and cryptic, and need not be shown to the user. It should be regarded the same way as the IP address and port number: it is another component of the addressing information required to identify a specific instance of a service being offered by some piece of hardware. Similarly, a file server may have multiple volumes, each identified by its own volume name. A web server typically has multiple pages, each identified by its own URL. In these cases, the necessary additional data is stored in a TXT record with the same name as the SRV record. The specific nature of that additional data, and how it is to be used, is service-dependent, but the overall syntax of the data in the TXT record is standardized, as described below.

Every DNS-SD service MUST have a TXT record in addition to its SRV record, with the same name, even if the service has no additional data to store and the TXT record contains no more than a single zero byte. This allows a service to have explicit control over the Time to Live (TTL) of its (empty) TXT record, rather than using the default negative caching TTL, which would otherwise be used for a "no error no answer" DNS response.

Note that this requirement for a mandatory TXT record applies exclusively to DNS-SD service advertising, i.e., services advertised using the PTR+SRV+TXT convention specified in this document. It is not a requirement of SRV records in general. The DNS SRV record datatype [RFC2782] may still be used in other contexts without any requirement for accompanying PTR and TXT records.

6.1. General Format Rules for DNS TXT Records

A DNS TXT record can be up to 65535 (0xFFFF) bytes long. The total length is indicated by the length given in the resource record header in the DNS message. There is no way to tell directly from the data alone how long it is (e.g., there is no length count at the start, or terminating NULL byte at the end).

Note that when using Multicast DNS [RFC6762] the maximum packet size is 9000 bytes, including the IP header, UDP header, and DNS message header, which imposes an upper limit on the size of TXT records of about 8900 bytes. In practice the maximum sensible size of a DNS-SD TXT record is smaller even than this, typically at most a few hundred bytes, as described below in Section 6.2.

The format of the data within a DNS TXT record is one or more strings, packed together in memory without any intervening gaps or padding bytes for word alignment.

The format of each constituent string within the DNS TXT record is a single length byte, followed by 0-255 bytes of text data.

These format rules for TXT records are defined in Section 3.3.14 of the DNS specification [RFC1035] and are not specific to DNS-SD. DNS-SD specifies additional rules for what data should be stored in those constituent strings when used for DNS-SD service advertising, i.e., when used to describe services advertised using the PTR+SRV+TXT convention specified in this document.

An empty TXT record containing zero strings is not allowed [RFC1035]. DNS-SD implementations MUST NOT emit empty TXT records. DNS-SD clients MUST treat the following as equivalent:

- A TXT record containing a single zero byte.
  (i.e., a single empty string.)

- An empty (zero-length) TXT record.
  (This is not strictly legal, but should one be received, it should be interpreted as the same as a single empty string.)

- No TXT record.
  (i.e., an NXDOMAIN or no-error-no-answer response.)

6.2. DNS-SD TXT Record Size

The total size of a typical DNS-SD TXT record is intended to be small -- 200 bytes or less.

In cases where more data is justified (e.g., LPR printing [BJP]), keeping the total size under 400 bytes should allow it to fit in a single 512-byte DNS message [RFC1035].

In extreme cases where even this is not enough, keeping the size of the TXT record under 1300 bytes should allow it to fit in a single 1500-byte Ethernet packet.

Using TXT records larger than 1300 bytes is NOT RECOMMENDED at this time.

Note that some Ethernet hardware vendors offer chipsets with Multicast DNS [RFC6762] offload, so that computers can sleep and still be discoverable on the network. Early versions of such chipsets were sometimes quite limited: for example, some were

(unwisely) limited to handling TXT records no larger than 256 bytes (which meant that LPR printer services with larger TXT records did not work). Developers should be aware of this real-world limitation, and should understand that even hardware which is otherwise perfectly capable may have low-power and sleep modes that are more limited.

6.3. DNS TXT Record Format Rules for Use in DNS-SD

DNS-SD uses DNS TXT records to store arbitrary key/value pairs conveying additional information about the named service. Each key/value pair is encoded as its own constituent string within the DNS TXT record, in the form "key=value" (without the quotation marks). Everything up to the first '=' character is the key (Section 6.4). Everything after the first '=' character to the end of the string (including subsequent '=' characters, if any) is the value (Section 6.5). No quotation marks are required around the value, even if it contains spaces, '=' characters, or other punctuation marks. Each author defining a DNS-SD profile for discovering instances of a particular type of service should define the base set of key/value attributes that are valid for that type of service.

Using this standardized key/value syntax within the TXT record makes it easier for these base definitions to be expanded later by defining additional named attributes. If an implementation sees unknown keys in a service TXT record, it MUST silently ignore them.

The target host name and TCP (or UDP) port number of the service are given in the SRV record. This information -- target host name and port number -- MUST NOT be duplicated using key/value attributes in the TXT record.

The intention of DNS-SD TXT records is to convey a small amount of useful additional information about a service. Ideally, it should not be necessary for a client to retrieve this additional information before it can usefully establish a connection to the service. For a well-designed application protocol, even if there is no information at all in the TXT record, it should be possible, knowing only the host name, port number, and protocol being used, to communicate with that listening process and then perform version- or feature-negotiation to determine any further options or capabilities of the service instance. For example, when connecting to an AFP (Apple Filing Protocol) server [AFP] over TCP, the client enters into a protocol exchange with the server to determine which version of AFP the server implements and which optional features or capabilities (if any) are available.

For protocols designed with adequate in-band version- and feature-negotiation, any information in the TXT record should be viewed as a

performance optimization -- when a client discovers many instances of a service, the TXT record allows the client to know some rudimentary information about each instance without having to open a TCP connection to each one and interrogate every service instance separately. Care should be taken when doing this to ensure that the information in the TXT record is in agreement with the information that would be retrieved by a client connecting over TCP.

There are legacy protocols that provide no feature negotiation capability, and in these cases it may be useful to convey necessary information in the TXT record. For example, when printing using LPR [RFC1179], the LPR protocol provides no way for the client to determine whether a particular printer accepts PostScript, what version of PostScript, etc. In this case it is appropriate to embed this information in the TXT record [BJP], because the alternative would be worse -- passing around written instructions to the users, arcane manual configuration of "/etc/printcap" files, etc.

The engineering decision about what keys to define for the TXT record needs to be decided on a case-by-case basis for each service type. For some service types it is appropriate to communicate information via the TXT record as well as (or instead of) via in-band communication in the application protocol.

6.4. Rules for Keys in DNS-SD Key/Value Pairs

The key MUST be at least one character. DNS-SD TXT record strings beginning with an '=' character (i.e., the key is missing) MUST be silently ignored.

The key SHOULD be no more than nine characters long. This is because it is beneficial to keep packet sizes small for the sake of network efficiency. When using DNS-SD in conjunction with Multicast DNS [RFC6762] this is important because multicast traffic is especially expensive on 802.11 wireless networks [IEEEW], but even when using conventional Unicast DNS, keeping the TXT records small helps improve the chance that responses will fit within the original DNS 512-byte size limit [RFC1035]. Also, each constituent string of a DNS TXT record is limited to 255 bytes, so excessively long keys reduce the space available for that key's values.

The keys in key/value pairs can be as short as a single character. A key name needs only to be unique and unambiguous within the context of the service type for which it is defined. A key name is intended solely to be a machine-readable identifier, not a human-readable essay giving detailed discussion of the purpose of a parameter, with a URL for a web page giving yet more details of the specification. For ease of development and debugging, it can be valuable to use key

names that are mnemonic textual names, but excessively verbose keys are wasteful and inefficient, hence the recommendation to keep them to nine characters or fewer.

The characters of a key MUST be printable US-ASCII values (0x20-0x7E) [RFC20], excluding '=' (0x3D).

Spaces in the key are significant, whether leading, trailing, or in the middle -- so don't include any spaces unless you really intend that.

Case is ignored when interpreting a key, so "papersize=A4", "PAPERSIZE=A4", and "Papersize=A4" are all identical.

If there is no '=' in a DNS-SD TXT record string, then it is a boolean attribute, simply identified as being present, with no value.

A given key SHOULD NOT appear more than once in a TXT record. The reason for this simplifying rule is to facilitate the creation of client libraries that parse the TXT record into an internal data structure (such as a hash table or dictionary object that maps from keys to values) and then make that abstraction available to client code. The rule that a given key may not appear more than once simplifies these abstractions because they aren't required to support the case of returning more than one value for a given key.

If a client receives a TXT record containing the same key more than once, then the client MUST silently ignore all but the first occurrence of that attribute. For client implementations that process a DNS-SD TXT record from start to end, placing key/value pairs into a hash table using the key as the hash table key, this means that if the implementation attempts to add a new key/value pair into the table and finds an entry with the same key already present, then the new entry being added should be silently discarded instead. Client implementations that retrieve key/value pairs by searching the TXT record for the requested key should search the TXT record from the start and simply return the first matching key they find.

When examining a TXT record for a given key, there are therefore four categories of results that may be returned:

* Attribute not present (Absent)

* Attribute present, with no value
  (e.g., "passreq" -- password required for this service)

* Attribute present, with empty value
  (e.g., "PlugIns=" -- the server supports plugins, but none are presently installed)

* Attribute present, with non-empty value
  (e.g., "PlugIns=JPEG,MPEG2,MPEG4")

Each author defining a DNS-SD profile for discovering instances of a particular type of service should define the interpretation of these different kinds of result. For example, for some keys, there may be a natural true/false boolean interpretation:

* Absent implies 'false'
* Present implies 'true'

For other keys it may be sensible to define other semantics, such as value/no-value/unknown:

* Present with value implies that value.
  (e.g., "Color=4" for a four-color ink-jet printer
  or "Color=6" for a six-color ink-jet printer)

* Present with empty value implies 'false'.
  (e.g., not a color printer)

* Absent implies 'Unknown'.
  (e.g., a print server connected to some unknown printer where the print server doesn't actually know if the printer does color or not -- which gives a very bad user experience and should be avoided wherever possible)

Note that this is a hypothetical example, not an example of actual key/value keys used by DNS-SD network printers, which are documented in the "Bonjour Printing Specification" [BJP].

6.5. Rules for Values in DNS-SD Key/Value Pairs

If there is an '=' in a DNS-SD TXT record string, then everything after the first '=' to the end of the string is the value. The value can contain any eight-bit values including '='. The value MUST NOT

be enclosed in additional quotation marks or any similar punctuation; any quotation marks, or leading or trailing spaces, are part of the value.

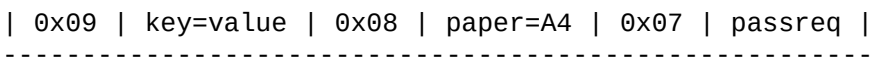
The value is opaque binary data. Often the value for a particular attribute will be US-ASCII [RFC20] or UTF-8 [RFC3629] text, but it is legal for a value to be any binary data.

Generic debugging tools should generally display all attribute values as a hex dump, with accompanying text alongside displaying the UTF-8 interpretation of those bytes, except for attributes where the debugging tool has embedded knowledge that the value is some other kind of data.

Authors defining DNS-SD profiles SHOULD NOT generically convert binary attribute data types into printable text using hexadecimal representation, Base-64 [RFC4648], or Unix-to-Unix (UU) encoding, merely for the sake of making the data appear to be printable text when seen in a generic debugging tool. Doing this simply bloats the size of the TXT record, without actually making the data any more understandable to someone looking at it in a generic debugging tool.

6.6. Example TXT Record

The TXT record below contains three syntactically valid key/value strings. (The meaning of these key/value pairs, if any, would depend on the definitions pertaining to the service in question that is using them.)



6.7. Version Tag

It is recommended that authors defining DNS-SD profiles include an attribute of the form "txtvers=x", where "x" is a decimal version number in US-ASCII [RFC20] text (e.g., "txtvers=1" or "txtvers=8"), in their definition, and require it to be the first key/value pair in the TXT record. This information in the TXT record can be useful to help clients maintain backwards compatibility with older implementations if it becomes necessary to change or update the specification over time. Even if the profile author doesn't anticipate the need for any future incompatible changes, having a version number in the TXT record provides useful insurance should incompatible changes become unavoidable [RFC6709]. Clients SHOULD ignore TXT records with a txtvers number higher (or lower) than the version(s) they know how to interpret.

Note that the version number in the txtvers tag describes the version of the specification governing the defined keys and the meaning of those keys for that particular TXT record, not the version of the application protocol that will be used if the client subsequently decides to contact that service. Ideally, every DNS-SD TXT record specification starts at txtvers=1 and stays that way forever. Improvements can be made by defining new keys that older clients silently ignore. The only reason to increment the version number is if the old specification is subsequently found to be so horribly broken that there's no way to do a compatible forward revision, so the txtvers number has to be incremented to tell all the old clients they should just not even try to understand this new TXT record.

If there is a need to indicate which version number(s) of the application protocol the service implements, the recommended key for this is "protovers".

6.8. Service Instances with Multiple TXT Records

Generally speaking, every DNS-SD service instance has exactly one TXT record. However, it is possible for a particular protocol's DNS-SD advertising specification to state that it allows multiple TXT records. In this case, each TXT record describes a different variant of the same logical service, offered using the same underlying protocol on the same port, described by the same SRV record.

Having multiple TXT records to describe a single service instance is very rare, and to date, of the many hundreds of registered DNS-SD service types [SN], only one makes use of this capability, namely LPR printing [BJP]. This capability is used when a printer conceptually supports multiple logical queue names, where each different logical queue name implements a different page description language, such as 80-column monospaced plain text, seven-bit Adobe PostScript, eight-bit ("binary") PostScript, or some proprietary page description language. When multiple TXT records are used to describe multiple logical LPR queue names for the same underlying service, printers include two additional keys in each TXT record: 'qtotal', which specifies the total number of TXT records associated with this SRV record, and 'priority', which gives the printer's relative preference for this particular TXT record. Clients then select the most preferred TXT record that meets the client's needs [BJP]. The only reason multiple TXT records are used is because the LPR protocol lacks in-band feature-negotiation capabilities for the client and server to agree on a data representation for the print job, so this information has to be communicated out-of-band instead using the DNS-SD TXT records. Future protocol designs should not follow this bad example by mimicking this inadequacy of the LPR printing protocol.

7. Service Names

The <Service> portion of a Service Instance Name consists of a pair of DNS labels, following the convention already established for SRV records [RFC2782].

The first label of the pair is an underscore character followed by the Service Name [RFC6335]. The Service Name identifies what the service does and what application protocol it uses to do it.

For applications using TCP, the second label is "_tcp".

For applications using any transport protocol other than TCP, the second label is "_udp". This applies to all other transport protocols, including User Datagram Protocol (UDP), Stream Control Transmission Protocol (SCTP) [RFC4960], Datagram Congestion Control Protocol (DCCP) [RFC4340], Adobe's Real Time Media Flow Protocol (RTMFP), etc. In retrospect, perhaps the SRV specification should not have used the "_tcp" and "_udp" labels at all, and instead should have used a single label "_srv" to carve off a subdomain of DNS namespace for this use, but that specification is already published and deployed. At this point there is no benefit in changing established practice. While "_srv" might be aesthetically nicer than "_udp", it is not a user-visible string, and all that is required protocol-wise is (i) that it be a label that can form a DNS delegation point, and (ii) that it be short so that it does not take up too much space in the packet, and in this respect either "_udp" or "_srv" is equally good. Thus, it makes sense to use "_tcp" for TCP-based services and "_udp" for all other transport protocols -- which are in fact, in today's world, often encapsulated over UDP -- rather than defining a new subdomain for every new transport protocol.

Note that this usage of the "_udp" label for all protocols other than TCP applies exclusively to DNS-SD service advertising, i.e., services advertised using the PTR+SRV+TXT convention specified in this document. It is not a requirement of SRV records in general. Other specifications that are independent of DNS-SD and not intended to interoperate with DNS-SD records are not in any way constrained by how DNS-SD works just because they also use the DNS SRV record datatype [RFC2782]; they are free to specify their own naming conventions as appropriate.

The rules for Service Names [RFC6335] state that they may be no more than fifteen characters long (not counting the mandatory underscore), consisting of only letters, digits, and hyphens, must begin and end with a letter or digit, must not contain consecutive hyphens, and must contain at least one letter. The requirement to contain at least one letter is to disallow Service Names such as "80" or

"6000-6063", which could be misinterpreted as port numbers or port number ranges. While both uppercase and lowercase letters may be used for mnemonic clarity, case is ignored for comparison purposes, so the strings "HTTP" and "http" refer to the same service.

Wise selection of a Service Name is important, and the choice is not always as obvious as it may appear.

In many cases, the Service Name merely names and refers to the on-the-wire message format and semantics being used. FTP is "ftp", IPP printing is "ipp", and so on.

However, it is common to "borrow" an existing protocol and repurpose it for a new task. This is entirely sensible and sound engineering practice, but that doesn't mean that the new protocol is providing the same semantic service as the old one, even if it borrows the same message formats. For example, the network music sharing protocol implemented by iTunes on Macintosh and Windows is built upon "HTTP GET" commands. However, that does *not* mean that it is sensible or useful to try to access one of these music servers by connecting to it with a standard web browser. Consequently, the DNS-SD service advertised (and browsed for) by iTunes is "_daap._tcp" (Digital Audio Access Protocol), not "_http._tcp".

If iTunes were to advertise that it offered "_http._tcp" service, that would cause iTunes servers to appear in conventional web browsers (Safari, Camino, OmniWeb, Internet Explorer, Firefox, Chrome, etc.), which is of little use since an iTunes music library offers no HTML pages containing human-readable content that a web browser could display.

Equally, if iTunes were to browse for "_http._tcp" service, that would cause it to discover generic web servers, such as the embedded web servers in devices like printers, which is of little use since printers generally don't have much music to offer.

Analogously, Sun Microsystems's Network File System (NFS) is built on top of Sun Microsystems's Remote Procedure Call (Sun RPC) mechanism, but that doesn't mean it makes sense for an NFS server to advertise that it provides "Sun RPC" service. Likewise, Microsoft's Server Message Block (SMB) file service is built on top of Netbios running over IP, but that doesn't mean it makes sense for an SMB file server to advertise that it provides "Netbios-over-IP" service. The DNS-SD name of a service needs to encapsulate both the "what" (semantics) and the "how" (protocol implementation) of the service, since knowledge of both is necessary for a client to use the service meaningfully. Merely advertising that a service was built on top of Sun RPC is no use if the client has no idea what the service does.

Another common question is whether the service type advertised by iTunes should be "_daap._http._tcp." This would also be incorrect. Similarly, a protocol designer implementing a network service that happens to use the Simple Object Access Protocol [SOAP] should not feel compelled to have "_soap" appear somewhere in the Service Name. Part of the confusion here is that the presence of "_tcp" or "_udp" in the <Service> portion of a Service Instance Name has led people to assume that the visible structure of the <Service> should reflect the private internal structure of how the protocol was implemented. This is not correct. All that is required is that the service be identified by some unique opaque Service Name. Making the Service Name be English text that is at least marginally descriptive of what the service does may be convenient, but it is by no means essential.

7.1. Selective Instance Enumeration (Subtypes)

This document does not attempt to define a sophisticated (e.g., Turing complete, or even regular expression) query language for service discovery, nor do we believe one is necessary.

However, there are some limited circumstances where narrowing the set of results may be useful. For example, many network printers offer a web-based user interface, for management and administration, using HTML/HTTP. A web browser wanting to discover all advertised web pages issues a query for "_http._tcp.<Domain>". On the other hand, there are cases where users wish to manage printers specifically, not to discover web pages in general, and it is good accommodate this. In this case, we define the "_printer" subtype of "_http._tcp", and to discover only the subset of pages advertised as having that subtype property, the web browser issues a query for "_printer._sub._http._tcp.<Domain>".

The Safari web browser on Mac OS X 10.5 "Leopard" and later uses subtypes in this way. If an "_http._tcp" service is discovered both via "_printer._sub._http._tcp" browsing and via "_http._tcp" browsing then it is displayed in the "Printers" section of Safari's UI. If a service is discovered only via "_http._tcp" browsing then it is displayed in the "Webpages" section of Safari's UI. This can be seen by using the commands below on Mac OS X to advertise two "fake" services. The service instance "A web page" is displayed in the "Webpages" section of Safari's Bonjour list, while the instance "A printer's web page" is displayed in the "Printers" section.

```
dns-sd -R "A web page"           _http._tcp          local 100
dns-sd -R "A printer's web page" _http._tcp,_printer local 101
```

Note that the advertised web page's Service Instance Name is unchanged by the use of subtypes -- it is still something of the form

"The Server._http._tcp.example.com.", and the advertised web page is still discoverable using a standard browsing query for services of type "_http._tcp". The subdomain in which HTTP server SRV records are registered defines the namespace within which HTTP server names are unique. Additional subtypes (e.g., "_printer") of the basic service type (e.g., "_http._tcp") serve to allow clients to query for a narrower set of results, not to create more namespace.

Using DNS zone file syntax, the service instance "A web page" is advertised using one PTR record, while the instance "A printer's web page" is advertised using two: the primary service type and the additional subtype. Even though the "A printer's web page" service is advertised two different ways, both PTR records refer to the name of the same SRV+TXT record pair:

```
; One PTR record advertises "A web page"
_http._tcp.local. PTR A\032web\032page._http._tcp.local.

; Two different PTR records advertise "A printer's web page"
_http._tcp.local. PTR A\032printer's\032web\032page._http._tcp.local.
_printer._sub._http._tcp.local.
                  PTR A\032printer's\032web\032page._http._tcp.local.
```

Subtypes are appropriate when it is desirable for different kinds of client to be able to browse for services at two levels of granularity. In the example above, we describe two classes of HTTP clients: general web browsing clients that are interested in all web pages, and specific printer management tools that would like to discover only web UI pages advertised by printers. The set of HTTP servers on the network is the same in both cases; the difference is that some clients want to discover all of them, whereas other clients only want to find the subset of HTTP servers whose purpose is printer administration.

Subtypes are only appropriate in two-level scenarios such as this one, where some clients want to find the full set of services of a given type, and at the same time other clients only want to find some subset. Generally speaking, if there is no client that wants to find the entire set, then it's neither necessary nor desirable to use the subtype mechanism. If all clients are browsing for some particular subtype, and no client exists that browses for the parent type, then a new Service Name representing the logical service should be defined, and software should simply advertise and browse for that particular service type directly. In particular, just because a particular network service happens to be implemented in terms of some other underlying protocol, like HTTP, Sun RPC, or SOAP, doesn't mean that it's sensible for that service to be defined as a subtype of "_http", "_sunrpc", or "_soap". That would only be useful if there

were some class of client for which it is sensible to say, "I want to discover a service on the network, and I don't care what it does, as long as it does it using the SOAP XML RPC mechanism."

Subtype strings are not required to begin with an underscore, though they often do. As with the TXT record key/value pairs, the list of possible subtypes, if any (including whether some or all begin with an underscore) are defined and specified separately for each basic service type.

Subtype strings (e.g., "_printer" in the example above) may be constructed using arbitrary 8-bit data values. In many cases these data values may be UTF-8 [RFC3629] representations of text, or even (as in the example above) plain ASCII [RFC20], but they do not have to be. Note, however, that even when using arbitrary 8-bit data for subtype strings, DNS name comparisons are still case-insensitive, so (for example) the byte values 0x41 and 0x61 will be considered equivalent for subtype comparison purposes.

7.2. Service Name Length Limits

As specified above, Service Names are allowed to be no more than fifteen characters long. The reason for this limit is to conserve bytes in the domain name for use both by the network administrator (choosing service domain names) and by the end user (choosing instance names).

A fully qualified domain name may be up to 255 bytes long, plus one byte for the final terminating root label at the end. Domain names used by DNS-SD take the following forms:

```
                <sn>._tcp . <servicedomain> . <parentdomain>.
   <Instance> . <sn>._tcp . <servicedomain> . <parentdomain>.
   <sub>._sub . <sn>._tcp . <servicedomain> . <parentdomain>.
```

The first example shows the name used for PTR queries. The second shows a Service Instance Name, i.e., the name of the service's SRV and TXT records. The third shows a subtype browsing name, i.e., the name of a PTR record pointing to a Service Instance Name (see Section 7.1, "Selective Instance Enumeration").

The Service Name <sn> may be up to 15 bytes, plus the underscore and length byte, making a total of 17. Including the "_udp" or "_tcp" and its length byte, this makes 22 bytes.

The instance name <Instance> may be up to 63 bytes. Including the length byte used by the DNS format when the name is stored in a packet, that makes 64 bytes.

When using subtypes, the subtype identifier is allowed to be up to 63 bytes, plus the length byte, making 64. Including the "_sub" and its length byte, this makes 69 bytes.

Typically, DNS-SD service records are placed into subdomains of their own beneath a company's existing domain name. Since these subdomains are intended to be accessed through graphical user interfaces, not typed on a command line, they are frequently long and descriptive. Including the length byte, the user-visible service domain may be up to 64 bytes.

Of our available 255 bytes, we have now accounted for 69+22+64 = 155 bytes. This leaves 100 bytes to accommodate the organization's existing domain name <parentdomain>. When used with Multicast DNS, <parentdomain> is "local.", which easily fits. When used with parent domains of 100 bytes or less, the full functionality of DNS-SD is available without restriction. When used with parent domains longer than 100 bytes, the protocol risks exceeding the maximum possible length of domain names, causing failures. In this case, careful choice of short <servicedomain> names can help avoid overflows. If the <servicedomain> and <parentdomain> are too long, then service instances with long instance names will not be discoverable or resolvable, and applications making use of long subtype names may fail.

Because of this constraint, we choose to limit Service Names to 15 characters or less. Allowing more characters would not increase the expressive power of the protocol and would needlessly reduce the maximum <parentdomain> length that may be safely used.

Note that <Instance> name lengths affect the maximum number of services of a given type that can be discovered in a given <servicedomain>. The largest Unicast DNS response than can be sent (typically using TCP, not UDP) is 64 kB. Using DNS name compression, a Service Instance Enumeration PTR record requires 2 bytes for the (compressed) name, plus 10 bytes for type, class, ttl, and rdata length. The rdata of the PTR record requires up to 64 bytes for the <Instance> part of the name, plus 2 bytes for a name compression pointer to the common suffix, making a maximum of 78 bytes total. This means that using maximum-sized <Instance> names, up to 839 instances of a given service type can be discovered in a given <servicedomain>.

Multicast DNS aggregates response packets, so it does not have the same hard limit, but in practice it is also useful for up to a few hundred instances of a given service type, but probably not thousands.

However, displaying even 100 instances in a flat list is probably too many to be helpful to a typical user. If a network has more than 100 instances of a given service type, it's probably appropriate to divide those services into logical subdomains by building, by floor, by department, etc.

8. Flagship Naming

In some cases, there may be several network protocols available that all perform roughly the same logical function. For example, the printing world has the lineprinter (LPR) protocol [RFC1179] and the Internet Printing Protocol (IPP) [RFC2910], both of which cause printed sheets to be emitted from printers in much the same way. In addition, many printer vendors send their own proprietary page description language (PDL) data over a TCP connection to TCP port 9100, herein referred to generically as the "pdl-datastream" protocol. In an ideal world, we would have only one network printing protocol, and it would be sufficiently good that no one felt a compelling need to invent a different one. However, in practice, multiple legacy protocols do exist, and a service discovery protocol has to accommodate that.

Many printers implement all three printing protocols: LPR, IPP, and pdl-datastream. For the benefit of clients that may speak only one of those protocols, all three are advertised.

However, some clients may implement two, or all three of those printing protocols. When a client looks for all three service types on the network, it will find three distinct services -- an LPR service, an IPP service, and a pdl-datastream service -- all of which cause printed sheets to be emitted from the same physical printer.

In a case like this, where multiple protocols all perform effectively the same function, a client may browse for all the service types it supports and display all the discovered instance names in a single aggregated list. Where the same instance name is discovered more than once because that entity supports more than one service type (e.g. a single printer which implements multiple printing protocols) the duplicates should be suppressed and the name should appear only once in the list. When the user indicates their desire to print on a given named printer, the printing client is responsible for choosing which of the available protocols will best achieve the desired effect, without, for example, requiring the user to make a manual choice between LPR and IPP.

As described so far, this all works very well. However, consider the case of: some future printer that only supports IPP printing, and some other future printer that only supports pdl-datastream printing.

The namespaces for different service types are intentionally disjoint (it is acceptable and desirable to be able to have both a file server called "Sales Department" and a printer called "Sales Department"). However, it is not desirable, in the common case, to allow two different printers both to be called "Sales Department" merely because those two printers implement different printing protocols.

To help guard against this, when there are two or more network protocols that perform roughly the same logical function, one of the protocols is declared the "flagship" of the fleet of related protocols. Typically the flagship protocol is the oldest and/or best-known protocol of the set.

If a device does not implement the flagship protocol, then it instead creates a placeholder SRV record (priority=0, weight=0, port=0, target host = host name of device) with that name. If, when it attempts to create this SRV record, it finds that a record with the same name already exists, then it knows that this name is already taken by some other entity implementing at least one of the protocols from the fleet, and it must choose another. If no SRV record already exists, then the act of creating it stakes a claim to that name so that future devices in the same protocol fleet will detect a conflict when they try to use it.

Note: When used with Multicast DNS [RFC6762], the target host field of the placeholder SRV record MUST NOT be the empty root label. The SRV record needs to contain a real target host name in order for the Multicast DNS conflict detection rules to operate. If two different devices were to create placeholder SRV records both using a null target host name (just the root label), then the two SRV records would be seen to be in agreement, and no conflict would be detected.

By defining a common well-known flagship protocol for the class, future devices that may not even know about each other’s protocols establish a common ground where they can coordinate to verify uniqueness of names.

No PTR record is created advertising the presence of empty flagship SRV records, since they do not represent a real service being advertised, and hence are not (and should not be) discoverable via Service Instance Enumeration (browsing).

9. Service Type Enumeration

In general, a normal client is not interested in finding *every* service on the network, just the services that the client knows how to use.

However, for problem diagnosis and network management tools, it may be useful for network administrators to find the list of advertised service types on the network, even if those Service Names are just opaque identifiers and not particularly informative in isolation.

For this purpose, a special meta-query is defined. A DNS query for PTR records with the name "_services._dns-sd._udp.<Domain>" yields a set of PTR records, where the rdata of each PTR record is the two-label <Service> name, plus the same domain, e.g., "_http._tcp.<Domain>". Including the domain in the PTR rdata allows for slightly better name compression in Unicast DNS responses, but only the first two labels are relevant for the purposes of service type enumeration. These two-label service types can then be used to construct subsequent Service Instance Enumeration PTR queries, in this <Domain> or others, to discover instances of that service type.

10. Populating the DNS with Information

How a service's PTR, SRV, and TXT records make their way into the DNS is outside the scope of this document, but, for illustrative purposes, some examples are given here.

On some networks, the administrator might manually enter the records into the name server's configuration file.

A network monitoring tool could output a standard zone file to be read into a conventional DNS server. For example, a tool that can find networked PostScript laser printers using AppleTalk NBP could find the list of printers, communicate with each one to find its IP address, PostScript version, installed options, etc., and then write out a DNS zone file describing those printers and their capabilities using DNS resource records. That information would then be available to IP-only clients that implement DNS-SD but not AppleTalk NBP.

A printer manager device that has knowledge of printers on the network through some other management protocol could also output a zone file or use DNS Update [RFC2136] [RFC3007].

Alternatively, a printer manager device could implement enough of the DNS protocol that it is able to answer DNS queries directly, and Example Co.'s main DNS server could delegate the "_ipp._tcp.example.com." subdomain to the printer manager device.

IP printers could use Dynamic DNS Update [RFC2136] [RFC3007] to automatically register their own PTR, SRV, and TXT records with the DNS server.

Zeroconf printers answer Multicast DNS queries on the local link for their own PTR, SRV, and TXT names ending with ".local." [RFC6762].

11. Discovery of Browsing and Registration Domains (Domain Enumeration)

One of the motivations for DNS-based Service Discovery is to enable a visiting client (e.g., a Wi-Fi-equipped [IEEEW] laptop computer, tablet, or mobile telephone) arriving on a new network to discover what services are available on that network, without any manual configuration. The logic that discovering services without manual configuration is a good idea also dictates that discovering recommended registration and browsing domains without manual configuration is a similarly good idea.

This discovery is performed using DNS queries, using Unicast or Multicast DNS. Five special RR names are reserved for this purpose:

```
 b._dns-sd._udp.<domain>.
db._dns-sd._udp.<domain>.
 r._dns-sd._udp.<domain>.
dr._dns-sd._udp.<domain>.
lb._dns-sd._udp.<domain>.
```

By performing PTR queries for these names, a client can learn, respectively:

- A list of domains recommended for browsing.
- A single recommended default domain for browsing.
- A list of domains recommended for registering services using Dynamic Update.
- A single recommended default domain for registering services.
- The "legacy browsing" or "automatic browsing" domain(s). Sophisticated client applications that care to present choices of domain to the user use the answers learned from the previous four queries to discover the domains to present. In contrast, many current applications browse without specifying an explicit domain, allowing the operating system to automatically select an appropriate domain on their behalf. It is for this class of application that the "automatic browsing" query is provided, to

allow the network administrator to communicate to the client operating systems which domain(s) should be used automatically for these applications.

These domains are purely advisory. The client or user is free to register services and/or browse in any domains. The purpose of these special queries is to allow software to create a user interface that displays a useful list of suggested choices to the user, from which the user may make an informed selection, or ignore the offered suggestions and manually enter their own choice.

The <domain> part of the Domain Enumeration query name may be "local." (meaning "perform the query using link-local multicast") or it may be learned through some other mechanism, such as the DHCP "Domain" option (option code 15) [RFC2132], the DHCP "Domain Search" option (option code 119) [RFC3397], or IPv6 Router Advertisement Options [RFC6106].

The <domain> part of the query name may also be derived a different way, from the host's IP address. The host takes its IP address and calculates the logical AND of that address and its subnet mask, to derive the 'base' address of the subnet (the 'network address' of that subnet, or, equivalently, the IP address of the 'all-zero' host address on that subnet). It then constructs the conventional DNS "reverse mapping" name corresponding to that base address, and uses that as the <domain> part of the name for the queries described above. For example, if a host has the address 192.168.12.34, with the subnet mask 255.255.0.0, then the 'base' address of the subnet is 192.168.0.0, and to discover the recommended automatic browsing domain(s) for devices on this subnet, the host issues a DNS PTR query for the name "lb._dns-sd._udp.0.0.168.192.in-addr.arpa."

Equivalent address-derived Domain Enumeration queries should also be done for the host's IPv6 address(es).

Address-derived Domain Enumeration queries SHOULD NOT be done for IPv4 link-local addresses [RFC3927] or IPv6 link-local addresses [RFC4862].

Sophisticated clients may perform Domain Enumeration queries both in "local." and in one or more unicast domains, using both name-derived and address-derived queries, and then present the user with an combined result, aggregating the information received from all sources.

12. DNS Additional Record Generation

DNS has an efficiency feature whereby a DNS server may place additional records in the additional section of the DNS message. These additional records are records that the client did not explicitly request, but the server has reasonable grounds to expect that the client might request them shortly, so including them can save the client from having to issue additional queries.

This section recommends which additional records SHOULD be generated to improve network efficiency, for both Unicast and Multicast DNS-SD responses.

Note that while servers SHOULD add these additional records for efficiency purposes, as with all DNS additional records, it is the client's responsibility to determine whether or not to trust them.

Generally speaking, stub resolvers that talk to a single recursive name server for all their queries will trust all records they receive from that recursive name server (whom else would they ask?). Recursive name servers that talk to multiple authoritative name servers should verify that any records they receive from a given authoritative name server are "in bailiwick" for that server, and ignore them if not.

Clients MUST be capable of functioning correctly with DNS servers (and Multicast DNS Responders) that fail to generate these additional records automatically, by issuing subsequent queries for any further record(s) they require. The additional-record generation rules in this section are RECOMMENDED for improving network efficiency, but are not required for correctness.

12.1. PTR Records

When including a DNS-SD Service Instance Enumeration or Selective Instance Enumeration (subtype) PTR record in a response packet, the server/responder SHOULD include the following additional records:

- The SRV record(s) named in the PTR rdata.
- The TXT record(s) named in the PTR rdata.
- All address records (type "A" and "AAAA") named in the SRV rdata.

12.2. SRV Records

When including an SRV record in a response packet, the server/responder SHOULD include the following additional records:

- All address records (type "A" and "AAAA") named in the SRV rdata.

12.3. TXT Records

When including a TXT record in a response packet, no additional records are required.

12.4. Other Record Types

In response to address queries, or other record types, no new additional records are recommended by this document.

13. Working Examples

The following examples were prepared using standard unmodified nslookup and standard unmodified BIND running on GNU/Linux.

Note: In real products, this information is obtained and presented to the user using graphical network browser software, not command-line tools. However, if you wish, you can try these examples for yourself as you read along, using the nslookup command already available on most Unix machines.

13.1. What web pages are being advertised from dns-sd.org?

```
nslookup -q=ptr _http._tcp.dns-sd.org.
_http._tcp.dns-sd.org
             name = Zeroconf._http._tcp.dns-sd.org
_http._tcp.dns-sd.org
             name = Multicast\032DNS._http._tcp.dns-sd.org
_http._tcp.dns-sd.org
             name = Service\032Discovery._http._tcp.dns-sd.org
_http._tcp.dns-sd.org
             name = Stuart's\032Printer._http._tcp.dns-sd.org
```

Answer: There are four, called "Zeroconf", "Multicast DNS", "Service Discovery", and "Stuart's Printer".

Note that nslookup escapes spaces as "\032" for display purposes, but a graphical DNS-SD browser should not.

13.2. What printer-configuration web pages are there?

```
nslookup -q=ptr _printer._sub._http._tcp.dns-sd.org.
_printer._sub._http._tcp.dns-sd.org
             name = Stuart's\032Printer._http._tcp.dns-sd.org
```

Answer: "Stuart's Printer" is the web configuration UI of a network printer.

13.3. How do I access the web page called "Service Discovery"?

```
nslookup -q=any "Service\032Discovery._http._tcp.dns-sd.org."
Service\032Discovery._http._tcp.dns-sd.org
               priority = 0, weight = 0, port = 80, host = dns-sd.org
Service\032Discovery._http._tcp.dns-sd.org
               text = "txtvers=1" "path=/"
dns-sd.org     nameserver = ns1.dns-sd.org
dns-sd.org     internet address = 64.142.82.154
ns1.dns-sd.org internet address = 64.142.82.152
```

Answer: You need to connect to dns-sd.org port 80, path "/".
The address for dns-sd.org is also given (64.142.82.154).

14. IPv6 Considerations

IPv6 has only minor differences from IPv4.

The address of the SRV record's target host is given by the appropriate IPv6 "AAAA" address records instead of (or in addition to) IPv4 "A" records.

Address-based Domain Enumeration queries are performed using names under the IPv6 reverse-mapping tree, which is different from the IPv4 reverse-mapping tree and has longer names in it.

15. Security Considerations

Since DNS-SD is just a specification for how to name and use records in the existing DNS system, it has no specific additional security requirements over and above those that already apply to DNS queries and DNS updates.

For DNS queries, DNS Security Extensions (DNSSEC) [RFC4033] should be used where the authenticity of information is important.

For DNS updates, secure updates [RFC2136] [RFC3007] should generally be used to control which clients have permission to update DNS records.

16. IANA Considerations

IANA manages the namespace of unique Service Names [RFC6335].

When a protocol service advertising specification includes subtypes, these should be documented in the protocol specification in question and/or in the "notes" field of the registration request sent to IANA. In the event that a new subtype becomes relevant after a protocol

specification has been published, this can be recorded by requesting that IANA add it to the "notes" field. For example, vendors of network printers advertise their embedded web servers using the subtype _printer. This allows printer management clients to browse for only printer-related web servers by browsing for the _printer subtype. While the existence of the _printer subtype of _http._tcp is not directly relevant to the HTTP protocol specification, it is useful to record this usage in the IANA registry to help avoid another community of developers inadvertently using the same subtype string for a different purpose. The namespace of possible subtypes is separate for each different service type. For example, the existence of the _printer subtype of _http._tcp does not imply that the _printer subtype is defined or has any meaning for any other service type.

When IANA records a Service Name registration, if the new application protocol is one that conceptually duplicates existing functionality of an older protocol, and the implementers desire the Flagship Naming behavior described in Section 8, then the registrant should request that IANA record the name of the flagship protocol in the "notes" field of the new registration. For example, the registrations for "ipp" and "pdl-datastream" both reference "printer" as the flagship name for this family of printing-related protocols.

17. Acknowledgments

The concepts described in this document have been explored, developed, and implemented with help from Ran Atkinson, Richard Brown, Freek Dijkstra, Ralph Droms, Erik Guttman, Pasi Sarolahti, Pekka Savola, Mark Townsley, Paul Vixie, Bill Woodcock, and others. Special thanks go to Bob Bradley, Josh Graessley, Scott Herscher, Rory McGuire, Roger Pantos, and Kiren Sekar for their significant contributions.

18. References

18.1. Normative References

[RFC20] Cerf, V., "ASCII format for network interchange", RFC 20, October 1969.

[RFC1033] Lottor, M., "Domain Administrators Operations Guide", RFC 1033, November 1987.

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, November 1987.

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

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

[RFC2782] Gulbrandsen, A., Vixie, P., and L. Esibov, "A DNS RR for specifying the location of services (DNS SRV)", RFC 2782, February 2000.

[RFC3492] Costello, A., "Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA)", RFC 3492, March 2003.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC3927] Cheshire, S., Aboba, B., and E. Guttman, "Dynamic Configuration of IPv4 Link-Local Addresses", RFC 3927, May 2005.

[RFC4862] Thomson, S., Narten, T., and T. Jinmei, "IPv6 Stateless Address Autoconfiguration", RFC 4862, September 2007.

[RFC5198] Klensin, J. and M. Padlipsky, "Unicode Format for Network Interchange", RFC 5198, March 2008.

[RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, August 2010.

[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, August 2011.

18.2. Informative References

[AFP] Mac OS X Developer Library, "Apple Filing Protocol Programming Guide", <http://developer.apple.com/documentation/Networking/Conceptual/AFP/>.

[BJ] Apple Bonjour Open Source Software, <http://developer.apple.com/bonjour/>.

[BJP] Bonjour Printing Specification, <https://developer.apple.com/bonjour/printing-specification/bonjourprinting-1.0.2.pdf>.

[IEEEW] IEEE 802 LAN/MAN Standards Committee, <http://standards.ieee.org/wireless/>.

[NIAS] Cheshire, S., "Discovering Named Instances of Abstract Services using DNS", Work in Progress, July 2001.

[NSD] "NsdManager | Android Developer", June 2012, <http://developer.android.com/reference/android/net/nsd/NsdManager.html>.

[RFC1179] McLaughlin, L., "Line printer daemon protocol", RFC 1179, August 1990.

[RFC2132] Alexander, S. and R. Droms, "DHCP Options and BOOTP Vendor Extensions", RFC 2132, March 1997.

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

[RFC2181] Elz, R. and R. Bush, "Clarifications to the DNS Specification", RFC 2181, July 1997.

[RFC2910] Herriot, R., Ed., Butler, S., Moore, P., Turner, R., and J. Wenn, "Internet Printing Protocol/1.1: Encoding and Transport", RFC 2910, September 2000.

[RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, September 2007.

[RFC3007] Wellington, B., "Secure Domain Name System (DNS) Dynamic Update", RFC 3007, November 2000.

[RFC4340] Kohler, E., Handley, M., and S. Floyd, "Datagram Congestion Control Protocol (DCCP)", RFC 4340, March 2006.

[RFC3397] Aboba, B. and S. Cheshire, "Dynamic Host Configuration Protocol (DHCP) Domain Search Option", RFC 3397, November 2002.

[RFC4033] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "DNS Security Introduction and Requirements", RFC 4033, March 2005.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[RFC4795] Aboba, B., Thaler, D., and L. Esibov, "Link-local Multicast Name Resolution (LLMNR)", RFC 4795, January 2007.

[RFC6106] Jeong, J., Park, S., Beloeil, L., and S. Madanapalli, "IPv6 Router Advertisement Options for DNS Configuration", RFC 6106, November 2010.

[RFC6281] Cheshire, S., Zhu, Z., Wakikawa, R., and L. Zhang, "Understanding Apple's Back to My Mac (BTMM) Service", RFC 6281, June 2011.

[RFC6709] Carpenter, B., Aboba, B., Ed., and S. Cheshire, "Design Considerations for Protocol Extensions", RFC 6709, September 2012.

[RFC6760] Cheshire, S. and M. Krochmal, "Requirements for a Protocol to Replace the AppleTalk Name Binding Protocol (NBP)", RFC 6760, February 2013.

[RFC6762] Cheshire, S. and M. Krochmal, "Multicast DNS", RFC 6762, February 2013.

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

[SOAP] Mitra, N., "SOAP Version 1.2 Part 0: Primer", W3C Proposed Recommendation 24 June 2003, <http://www.w3.org/TR/2003/REC-soap12-part0-20030624>.

[Unicode6] The Unicode Consortium. The Unicode Standard, Version 6.0.0, (Mountain View, CA: The Unicode Consortium, 2011. ISBN 978-1-936213-01-6) <http://www.unicode.org/versions/Unicode6.0.0/>.

[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. Rationale for Using DNS as a Basis for Service Discovery

Over the years, there have been many proposed ways to do network service discovery with IP, but none achieved ubiquity in the marketplace. Certainly none has achieved anything close to the ubiquity of today's deployment of DNS servers, clients, and other infrastructure.

The advantage of using DNS as the basis for service discovery is that it makes use of those existing servers, clients, protocols, infrastructure, and expertise. Existing network analyzer tools already know how to decode and display DNS packets for network debugging.

For ad hoc networks such as Zeroconf environments, peer-to-peer multicast protocols are appropriate. Using DNS-SD running over Multicast DNS [RFC6762] provides zero-configuration ad hoc service discovery, while maintaining the DNS-SD semantics and record types described here.

In larger networks, a high volume of enterprise-wide IP multicast traffic may not be desirable, so any credible service discovery protocol intended for larger networks has to provide some facility to aggregate registrations and lookups at a central server (or servers) instead of working exclusively using multicast. This requires some service discovery aggregation server software to be written, debugged, deployed, and maintained. This also requires some service discovery registration protocol to be implemented and deployed for clients to register with the central aggregation server. Virtually every company with an IP network already runs a DNS server, and DNS already has a dynamic registration protocol [RFC2136] [RFC3007]. Given that virtually every company already has to operate and maintain a DNS server anyway, it makes sense to take advantage of this expertise instead of also having to learn, operate, and maintain a different service registration server. It should be stressed again that using the same software and protocols doesn't necessarily mean using the same physical piece of hardware. The DNS-SD service discovery functions do not have to be provided by the same piece of hardware that is currently providing the company's DNS name service. The "_tcp.<Domain>" and "_udp.<Domain>" subdomains may be delegated to a different piece of hardware. However, even when the DNS-SD service is being provided by a different piece of hardware, it is still the same familiar DNS server software, with the same configuration file syntax, the same log file format, and so forth.

Service discovery needs to be able to provide appropriate security. DNS already has existing mechanisms for security [RFC4033].

In summary:

Service discovery requires a central aggregation server.
DNS already has one: a DNS server.

Service discovery requires a service registration protocol.
DNS already has one: DNS Dynamic Update.

Service discovery requires a query protocol.
DNS already has one: DNS queries.

Service discovery requires security mechanisms.
DNS already has security mechanisms: DNSSEC.

Service discovery requires a multicast mode for ad hoc networks.
Using DNS-SD in conjunction with Multicast DNS provides this,
using peer-to-peer multicast instead of a DNS server.

It makes more sense to use the existing software that every network needs already, instead of deploying an entire parallel system just for service discovery.

Appendix B. Ordering of Service Instance Name Components

There have been questions about why services are named using DNS Service Instance Names of the form:

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

instead of:

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

There are three reasons why it is beneficial to name service instances with the parent domain as the most-significant (rightmost) part of the name, then the abstract service type as the next-most significant, and then the specific instance name as the least-significant (leftmost) part of the name. These reasons are discussed below in Sections B.1, B.2, and B.3.

B.1. Semantic Structure

The facility being provided by browsing ("Service Instance Enumeration") is effectively enumerating the leaves of a tree structure. A given domain offers zero or more services. For each of those service types, there may be zero or more instances of that service.

The user knows what type of service they are seeking. (If they are running an FTP client, they are looking for FTP servers. If they have a document to print, they are looking for entities that speak some known printing protocol.) The user knows in which organizational or geographical domain they wish to search. (The user does not want a single flat list of every single printer on the planet, even if such a thing were possible.) What the user does not know in advance is whether the service they seek is offered in the given domain, or if so, the number of instances that are offered and the names of those instances.

Hence, having the instance names be the leaves of the tree is consistent with this semantic model.

Having the service types be the terminal leaves of the tree would imply that the user knows the domain name and the name of the service instance, but doesn't have any idea what the service does. We would argue that this is a less useful model.

B.2. Network Efficiency

When a DNS response contains multiple answers, name compression works more effectively if all the names contain a common suffix. If many answers in the packet have the same <Service> and <Domain>, then each occurrence of a Service Instance Name can be expressed using only the <Instance> part followed by a two-byte compression pointer referencing a previous appearance of "<Service>.<Domain>". This efficiency would not be possible if the <Service> component appeared first in each name.

B.3. Operational Flexibility

This name structure allows subdomains to be delegated along logical service boundaries. For example, the network administrator at Example Co. could choose to delegate the "_tcp.example.com." subdomain to a different machine, so that the machine handling service discovery doesn't have to be the machine that handles other day-to-day DNS operations. (It *can* be the same machine if the administrator so chooses, but the administrator is free to make that choice.) Furthermore, if the network administrator wishes to delegate all information related to IPP printers to a machine dedicated to that specific task, this is easily done by delegating the "_ipp._tcp.example.com." subdomain to the desired machine. It is also convenient to set security policies on a per-zone/per-subdomain basis. For example, the administrator may choose to enable DNS Dynamic Update [RFC2136] [RFC3007] for printers registering in the

"_ipp._tcp.example.com." subdomain, but not for other zones/subdomains. This easy flexibility would not exist if the <Service> component appeared first in each name.

Appendix C. What You See Is What You Get

Some service discovery protocols decouple the true service identifier from the name presented to the user. The true service identifier used by the protocol is an opaque unique identifier, often represented using a long string of hexadecimal digits, which should never be seen by the typical user. The name presented to the user is merely one of the decorative ephemeral attributes attached to this opaque identifier.

The problem with this approach is that it decouples user perception from network reality:

* What happens if there are two service instances, with different unique ids, but they have inadvertently been given the same user-visible name? If two instances appear in an on-screen list with the same name, how does the user know which is which?

* Suppose a printer breaks down, and the user replaces it with another printer of the same make and model, and configures the new printer with the exact same name as the one being replaced: "Stuart's Printer". Now, when the user tries to print, the on-screen print dialog tells them that their selected default printer is "Stuart's Printer". When they browse the network to see what is there, they see a printer called "Stuart's Printer", yet when the user tries to print, they are told that the printer "Stuart's Printer" can't be found. The hidden internal unique identifier that the software is trying to find on the network doesn't match the hidden internal unique identifier of the new printer, even though its apparent "name" and its logical purpose for being there are the same. To remedy this, the user typically has to delete the print queue they have created, and then create a new (apparently identical) queue for the new printer, so that the new queue will contain the right hidden internal unique identifier. Having all this hidden information that the user can't see makes for a confusing and frustrating user experience, and exposing long, ugly hexadecimal strings to the user and forcing them to understand what they mean is even worse.

* Suppose an existing printer is moved to a new department, and given a new name and a new function. Changing the user-visible name of that piece of hardware doesn't change its hidden internal unique identifier. Users who had previously created a print queue

for that printer will still be accessing the same hardware by its unique identifier, even though the logical service that used to be offered by that hardware has ceased to exist.

Solving these problems requires the user or administrator to be aware of the supposedly hidden unique identifier, and to set its value correctly as hardware is moved around, repurposed, or replaced, thereby contradicting the notion that it is a hidden identifier that human users never need to deal with. Requiring the user to understand this expert behind-the-scenes knowledge of what is *really* going on is just one more burden placed on the user when they are trying to diagnose why their computers and network devices are not working as expected.

These anomalies and counterintuitive behaviors can be eliminated by maintaining a tight bidirectional one-to-one mapping between what the user sees on the screen and what is really happening "behind the curtain". If something is configured incorrectly, then that is apparent in the familiar day-to-day user interface that everyone understands, not in some little-known, rarely used "expert" interface.

In summary: in DNS-SD the user-visible name is also the primary identifier for a service. If the user-visible name is changed, then conceptually the service being offered is a different logical service -- even though the hardware offering the service may have stayed the same. If the user-visible name doesn't change, then conceptually the service being offered is the same logical service -- even if the hardware offering the service is new hardware brought in to replace some old equipment.

There are certainly arguments on both sides of this debate. Nonetheless, the designers of any service discovery protocol have to make a choice between having the primary identifiers be hidden, or having them be visible, and these are the reasons that we chose to make them visible. We're not claiming that there are no disadvantages of having primary identifiers be visible. We considered both alternatives, and we believe that the few disadvantages of visible identifiers are far outweighed by the many problems caused by use of hidden identifiers.

Appendix D. Choice of Factory-Default Names

When a DNS-SD service is advertised using Multicast DNS [RFC6762], if there is already another service of the same type advertising with the same name then automatic name conflict resolution will occur. As described in the Multicast DNS specification [RFC6762], upon detecting a conflict, the service should:

1. Automatically select a new name (typically by appending or incrementing a digit at the end of the name),
2. Try advertising with the new name, and
3. Upon success, record the new name in persistent storage.

This renaming behavior is very important, because it is key to providing user-friendly instance names in the out-of-the-box factory-default configuration. Some product developers apparently have not realized this, because there are some products today where the factory-default name is distinctly unfriendly, containing random-looking strings of characters, such as the device's Ethernet address in hexadecimal. This is unnecessary and undesirable, because the point of the user-visible name is that it should be friendly and meaningful to human users. If the name is not unique on the local network then the protocol will remedy this as necessary. It is ironic that many of the devices with this design mistake are network printers, given that these same printers also simultaneously support AppleTalk-over-Ethernet, with nice user-friendly default names (and automatic conflict detection and renaming). Some examples of good factory-default names are:

```
Brother 5070N
Canon W2200
HP LaserJet 4600
Lexmark W840
Okidata C5300
Ricoh Aficio CL7100
Xerox Phaser 6200DX
```

To make the case for why adding long, ugly factory-unique serial numbers to the end of names is neither necessary nor desirable, consider the cases where the user has (a) only one network printer, (b) two network printers, and (c) many network printers.

(a) In the case where the user has only one network printer, a simple name like (to use a vendor-neutral example) "Printer" is more user-friendly than an ugly name like "Printer_0001E68C74FB". Appending ugly hexadecimal goop to the end of the name to make sure the name is unique is irrelevant to a user who only has one printer anyway.

(b) In the case where the user gets a second network printer, having the new printer detect that the name "Printer" is already in use and automatically name itself "Printer (2)" instead, provides a good user experience. For most users, remembering that the old printer is "Printer" and the new one is "Printer (2)" is easy and intuitive. Seeing a printer called "Printer_0001E68C74FB" and another called "Printer_00306EC3FD1C" is a lot less helpful.

(c) In the case of a network with ten network printers, seeing a list of ten names all of the form "Printer_xxxxxxxxxxxx" has effectively taken what was supposed to be a list of user-friendly rich-text names (supporting mixed case, spaces, punctuation, non-Roman characters, and other symbols) and turned it into just about the worst user interface imaginable: a list of incomprehensible random-looking strings of letters and digits. In a network with a lot of printers, it would be advisable for the people setting up the printers to take a moment to give each one a descriptive name, but in the event they don't, presenting the users with a list of sequentially numbered printers is a much more desirable default user experience than showing a list of raw Ethernet addresses.

Appendix E. Name Encodings in the Domain Name System

Although the original DNS specifications [RFC1033] [RFC1034] [RFC1035] recommend that host names contain only letters, digits, and hyphens (because of the limitations of the typing-based user interfaces of that era), Service Instance Names are not host names. Users generally access a service by selecting it from a list presented by a user interface, not by typing in its Service Instance Name. "Clarifications to the DNS Specification" [RFC2181] directly discusses the subject of allowable character set in Section 11 ("Name syntax"), and explicitly states that the traditional letters-digits-hyphens rule applies only to conventional host names:

> Occasionally it is assumed that the Domain Name System serves only the purpose of mapping Internet host names to data, and mapping Internet addresses to host names. This is not correct, the DNS is a general (if somewhat limited) hierarchical database, and can store almost any kind of data, for almost any purpose.
>
> The DNS itself places only one restriction on the particular labels that can be used to identify resource records. That one restriction relates to the length of the label and the full name. The length of any one label is limited to between 1 and 63 octets. A full domain name is limited to 255 octets (including the separators). The zero length full name is defined as representing the root of the DNS tree, and is typically written and displayed as ".". Those restrictions aside, any binary string whatever can be used as the label of any resource record. Similarly, any binary string can serve as the value of any record that includes a domain name as some or all of its value (SOA, NS, MX, PTR, CNAME, and any others that may be added). Implementations of the DNS protocols must not place any restrictions on the labels that can be used. In particular, DNS servers must not refuse to serve a zone because it contains labels that might not be acceptable to some DNS client programs.

Note that just because DNS-based Service Discovery supports arbitrary UTF-8-encoded names doesn't mean that any particular user or administrator is obliged to make use of that capability. Any user is free, if they wish, to continue naming their services using only letters, digits, and hyphens, with no spaces, capital letters, or other punctuation.

Appendix F. "Continuous Live Update" Browsing Model

Of particular concern in the design of DNS-SD, especially when used in conjunction with ad hoc Multicast DNS, is the dynamic nature of service discovery in a changing network environment. Other service discovery protocols seem to have been designed with an implicit unstated assumption that the usage model is:

(a) client software calls the service discovery API,
(b) service discovery code spends a few seconds getting a list of instances available at a particular moment in time, and then
(c) client software displays the list for the user to select from.

Superficially this usage model seems reasonable, but the problem is that it's too optimistic. It only considers the success case, where the software immediately finds the service instance the user is looking for.

In the case where the user is looking for (say) a particular printer, and that printer is not turned on or not connected, the user first has to attempt to remedy the problem, and then has to click a "refresh" button to retry the service discovery to find out whether they were successful. Because nothing happens instantaneously in networking, and packets can be lost, necessitating some number of retransmissions, a service discovery search is not instantaneous and typically takes a few seconds. As a result, a fairly typical user experience is:

(a) display an empty window,
(b) display some animation like a searchlight sweeping back and forth for ten seconds, and then
(c) at the end of the ten-second search, display a static list showing what was discovered.

Every time the user clicks the "refresh" button they have to endure another ten-second wait, and every time the discovered list is finally shown at the end of the ten-second wait, it's already beginning to get stale and out-of-date the moment it's displayed on the screen.

The service discovery user experience that the DNS-SD designers had in mind has some rather different properties:

1. Displaying the initial list of discovered services should be effectively instantaneous -- i.e., typically 0.1 seconds, not 10 seconds.

2. The list of discovered services should not be getting stale and out-of-date from the moment it's displayed. The list should be 'live' and should continue to update as new services are discovered. Because of the delays, packet losses, and retransmissions inherent in networking, it is to be expected that sometimes, after the initial list is displayed showing the majority of discovered services, a few remaining stragglers may continue to trickle in during the subsequent few seconds. Even after this stable list has been built and displayed, it should remain 'live' and should continue to update. At any future time, be it minutes, hours, or even days later, if a new service of the desired type is discovered, it should be displayed in the list automatically, without the user having to click a "refresh" button or take any other explicit action to update the display.

3. With users getting in the habit of leaving service discovery windows open, and expecting them to show a continuous 'live' view of current network reality, this gives us an additional requirement: deletion of stale services. When a service discovery list shows just a static snapshot at a moment in time, then the situation is simple: either a service was discovered and appears in the list, or it was not and does not. However, when our list is live and updates continuously with the discovery of new services, then this implies the corollary: when a service goes away, it needs to *disappear* from the service discovery list. Otherwise, the service discovery list would simply grow monotonically over time, accreting stale data, and would require a periodic "refresh" (or complete dismissal and recreation) to restore correct display.

4. Another consequence of users leaving service discovery windows open for extended periods of time is that these windows should update not only in response to services coming and going, but also in response to changes in configuration and connectivity of the client machine itself. For example, if a user opens a service discovery window when the client machine has no network connectivity, then the window will typically appear empty, with no discovered services. When the user connects an Ethernet cable or joins an 802.11 [IEEEW] wireless network the window should then automatically populate with discovered services, without requiring any explicit user action. If the user disconnects the Ethernet cable or turns off 802.11 wireless then all the services discovered via that network interface should automatically disappear. If the user switches from one 802.11 wireless access point to another, the service discovery window should automatically update to remove all the services discovered via the old wireless access point, and add all the services discovered via the new one.

Appendix G. Deployment History

In July 1997, in an email to the net-thinkers@thumper.vmeng.com mailing list, Stuart Cheshire first proposed the idea of running the AppleTalk Name Binding Protocol [RFC6760] over IP. As a result of this and related IETF discussions, the IETF Zeroconf working group was chartered September 1999. After various working group discussions and other informal IETF discussions, several Internet-Drafts were written that were loosely related to the general themes of DNS and multicast, but did not address the service discovery aspect of NBP.

In April 2000, Stuart Cheshire registered IPv4 multicast address 224.0.0.251 with IANA and began writing code to test and develop the idea of performing NBP-like service discovery using Multicast DNS, which was documented in a group of three Internet-Drafts:

- "Requirements for a Protocol to Replace the AppleTalk Name Binding Protocol (NBP)" [RFC6760] is an overview explaining the AppleTalk Name Binding Protocol, because many in the IETF community had little first-hand experience using AppleTalk, and confusion in the IETF community about what AppleTalk NBP did was causing confusion about what would be required in an IP-based replacement.

- "Discovering Named Instances of Abstract Services using DNS" [NIAS], which later became this document, proposed a way to perform NBP-like service discovery using DNS-compatible names and record types.

- "Multicast DNS" [RFC6762] specifies a way to transport those DNS-compatible queries and responses using IP multicast, for zero-configuration environments where no conventional Unicast DNS server was available.

In 2001, an update to Mac OS 9 added resolver library support for host name lookup using Multicast DNS. If the user typed a name such as "MyPrinter.local." into any piece of networking software that used the standard Mac OS 9 name lookup APIs, then those name lookup APIs would recognize the name as a dot-local name and query for it by sending simple one-shot Multicast DNS queries to 224.0.0.251:5353. This enabled the user to, for example, enter the name "MyPrinter.local." into their web browser in order to view a printer's status and configuration web page, or enter the name "MyPrinter.local." into the printer setup utility to create a print queue for printing documents on that printer.

Multicast DNS responder software, with full service discovery, first began shipping to end users in volume with the launch of Mac OS X 10.2 "Jaguar" in August 2002, and network printer makers (who had historically supported AppleTalk in their network printers and were receptive to IP-based technologies that could offer them similar ease-of-use) started adopting Multicast DNS shortly thereafter.

In September 2002, Apple released the source code for the mDNSResponder daemon as Open Source under Apple's standard Apple Public Source License (APSL).

Multicast DNS responder software became available for Microsoft Windows users in June 2004 with the launch of Apple's "Rendezvous for Windows" (now "Bonjour for Windows"), both in executable form (a downloadable installer for end users) and as Open Source (one of the supported platforms within Apple's body of cross-platform code in the publicly accessible mDNSResponder CVS source code repository) [BJ].

In August 2006, Apple re-licensed the cross-platform mDNSResponder source code under the Apache License, Version 2.0.

In addition to desktop and laptop computers running Mac OS X and Microsoft Windows, Multicast DNS is now implemented in a wide range of hardware devices, such as Apple's "AirPort" wireless base stations, iPhone and iPad, and in home gateways from other vendors, network printers, network cameras, TiVo DVRs, etc.

The Open Source community has produced many independent implementations of Multicast DNS, some in C like Apple's mDNSResponder daemon, and others in a variety of different languages including Java, Python, Perl, and C#/Mono.

In January 2007, the IETF published the Informational RFC "Link-Local Multicast Name Resolution (LLMNR)" [RFC4795], which is substantially similar to Multicast DNS, but incompatible in some small but important ways. In particular, the LLMNR design explicitly excluded support for service discovery, which made it an unsuitable candidate for a protocol to replace AppleTalk NBP [RFC6760].

While the original focus of Multicast DNS and DNS-Based Service Discovery was for zero-configuration environments without a conventional Unicast DNS server, DNS-Based Service Discovery also works using Unicast DNS servers, using DNS Update [RFC2136] [RFC3007] to create service discovery records and standard DNS queries to query for them. Apple's Back to My Mac service, launched with Mac OS X 10.5 "Leopard" in October 2007, uses DNS-Based Service Discovery over Unicast DNS [RFC6281].

In June 2012, Google’s Android operating system added native support for DNS-SD and Multicast DNS with the android.net.nsd.NsdManager class in Android 4.1 "Jelly Bean" (API Level 16) [NSD].

Authors’ Addresses

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

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

Marc Krochmal
Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
USA

Phone: +1 408 974 4368
EMail: marc@apple.com