Document: draft-cheshire-dnsext-dns-sd-04.txt
Internet-Draft
Category: Standards Track
Expires 10th February 2007

Stuart Cheshire
Marc Krochmal
Apple Computer, Inc.

10th August 2006

DNS-Based Service Discovery

<draft-cheshire-dnsext-dns-sd-04.txt>

Status of this Memo

Abstract

This document describes a convention for naming and structuring DNS resource records. Given a type of service that a client is looking for, and a domain in which the client is looking for that service, this convention allows clients to discover a list of named instances of that desired service, using only standard DNS queries. In short, this is referred to as DNS-based Service Discovery, or DNS-SD.

Table of Contents

1. Introduction

This document describes a convention for naming and structuring DNS resource records. Given a type of service that a client is looking for, and a domain in which the client is looking for that service, this convention allows clients to discover a list of named instances of a that desired service, using only standard DNS queries. In short, this 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 simply proposes a convention for how existing resource record types can be named and structured to facilitate service discovery.

This proposal is entirely compatible with today's existing unicast DNS server and client software.

Note that the 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 (the service we traditionally think of when we say "DNS"). By delegating the "_tcp" subdomain, 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 things that makes DNS-SD so compelling.

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.

This proposal is also compatible with (but not dependent on) the proposal outlined in "Multicast DNS" [mDNS].

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" [RFC 2119].

3. Design Goals

A good service discovery protocol needs to have many properties, three of which are mentioned below:

(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 Name 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 network browsing step 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 AppleTalk NBP" [NBP]. That document draws upon examples from two decades of operational experience with AppleTalk Name Binding Protocol to develop a list of universal requirements which are broadly applicable to any potential service discovery protocol.

4. Service Instance Enumeration

DNS SRV records [RFC 2782] 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 a list of all 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 RFC 2782.

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 IPP 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, in order to select the particular instance the user desires.

4.1 Structured 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).

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 the PTR lookup is akin to performing a listing of that directory to find all the files it contains. (Remember that domain names are expressed in reverse order compared to path names: An absolute path name is read from left to right, beginning with a leading slash on the left, and then the top level directory, then the next level directory, and so on. A fully-qualified domain name is read from right to left, beginning with the dot on the right -- the root label -- and then the top level domain to the left of that, and the second level domain to the left of that, and so on. 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 list of zero or more PTR records giving Service Instance Names of the form:

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

The <Instance> portion of the Service Instance Name is a single DNS label, containing arbitrary precomposed UTF-8-encoded text [RFC 3629]. It is a user-friendly name, meaning that it is allowed to contain any characters, without restriction, including spaces, upper case, lower case, punctuation -- including dots -- accented characters, non-roman text, and anything else that may be represented using UTF-8. DNS recommends guidelines for allowable characters for host names [RFC 1033][RFC 1034][RFC 1035], but Service Instance Names are not host names. Service Instance Names are not intended to ever be typed in by a normal user; the user selects a Service Instance Name by selecting it from a list of choices presented on the screen.

Note that just because this protocol 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.

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 UTF-8 encodings tend to be the more obscure ones, and tend to be the ones that convey greater meaning per character.

Note that any character in the commonly-used 16-bit Unicode space 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.

The <Service> portion of the Service Instance Name consists of a pair of DNS labels, following the established convention for SRV records [RFC 2782], namely: the first label of the pair is the Application Protocol Name, and the second label is either "_tcp" or "_udp", depending on the transport protocol used by the application.
More details are given in Section 7, "Application Protocol Names".

The <Domain> portion of the Service Instance Name specifies the DNS subdomain within which the service names are registered. It may be "local", meaning "link-local Multicast DNS" [mDNS], or it may be a conventional unicast DNS domain name, such as "apple.com.", "cs.stanford.edu.", or "eng.us.ibm.com." Because service names are not host names, they are not constrained by the usual rules for host names [RFC 1033][RFC 1034][RFC 1035], and rich-text service subdomains are allowed and encouraged, for example:

```
  Building 2, 1st Floor.apple.com.
  Building 2, 2nd Floor.apple.com.
  Building 2, 3rd Floor.apple.com.
  Building 2, 4th Floor.apple.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, Unicode Normalization Form C [UAX15]. In cases where the DNS server returns an NXDOMAIN error for the name in question, client software MAY choose to retry the query using "Punycode" [RFC 3492] encoding, if possible.

4.2 User Interface Presentation

The names resulting from the 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 user, these having been provided 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: 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 "Selective Instance Enumeration") the <Service> of the discovered instance my not be exactly the same as the <Service> that was requested.

Having chosen 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 the <Instance>, <Service> and <Domain> portions are internally concatenated together into a single string, then care must be taken with the <Instance> portion, since it is allowed to contain any characters, including dots.

Any dots in the <Instance> portion should be escaped by preceding them with a backslash ("." becomes "\."). Likewise, any backslashes in the <Instance> portion should also be escaped by preceding them with a backslash ("\" becomes "\\"). Having done this, the three components of the name may be safely concatenated. The backslashescaping allows literal dots in the name (escaped) to be distinguished from label-separator dots (not escaped).

The resulting concatenated string may be safely passed to standard DNS APIs like res_query(), which will interpret the string correctly provided it has been escaped correctly, as described here.

4.4 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 id, often represented using a long string of hexadecimal digits, and should never be seen by the typical user. The name presented to the user is merely one of the ephemeral attributes attached to this opaque identifier.

The problem with this approach is that it decouples user perception from 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 id that the software is trying to find on the network doesn't match the hidden internal unique id 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 id. 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

id. Users who had previously created print queues for that printer will still be accessing the same hardware by its unique id, even though the logical service that used to be offered by that hardware has ceased to exist.

To solve these problems requires the user or administrator to be aware of the supposedly hidden unique id, 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 counter-intuitive 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: The user-visible name is 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 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 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.

4.5 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:

4.5.1. 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, how many instances 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 already knows 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.

4.5.2. 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.

4.5.3. 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 same as the machine handling other day-to-day DNS operations. (It *can* be the same machine if the administrator so chooses, but the point is that 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 [RFC 2136] [RFC 3007] 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.

5. Service Name Resolution

Given a particular Service Instance Name, when a client needs to contact that service, it sends a DNS query for the SRV record of that name.

The result of the DNS query is a SRV record giving the port number and target host where the service may be found.

The use of SRV records is very important. There are only 65535 TCP port numbers available. These port numbers are being allocated one-per-application-protocol at an alarming rate. Some protocols like the X Window System have a block of 64 TCP ports allocated (6000-6063). If we start allocating blocks of 64 TCP ports at a time, we will run out even faster. Using a different TCP port for each different instance of a given service on a given machine is entirely sensible, but allocating large static ranges, as was done for X, is a very inefficient way to manage a limited resource. On any given host, most TCP ports are reserved for services that will never run on that particular host. 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 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 some environments there may be no compelling reason to assign managed names to every host, since every available service is accessible by name anyway, as a first-class entity in its own right. However, the DNS packet format and record format still require a host name to link the target host referenced in the SRV record to the address records giving the IPv4 and/or IPv6 addresses for that hardware. In the case where no natural host name is available, the SRV record may give its own name as the name of the target host, and then the requisite address records may be attached to that same name.

It is perfectly permissible for a single name in the DNS hierarchy to have multiple records of different type attached. (The only restriction being that a given name may not have both a CNAME record and other records at the same time.)

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 properly identify the service. For example, printing via the LPR protocol often specifies a queue name. 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 one 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 same name, even if the service has no additional data to store and the TXT record contains no more than a single zero byte.

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 [mDNS] the maximum packet size is 9000 bytes, which imposes an upper limit on the size of TXT records of about 8800 bytes.)

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 are defined in Section 3.3.14 of RFC 1035, and are not specific to DNS-SD. DNS-SD simply specifies a usage convention for what data should be stored in those constituent strings.

An empty TXT record containing zero strings is disallowed by RFC 1035. DNS-SD implementations MUST NOT emit empty TXT records. DNS-SD implementations receiving empty TXT records MUST treat them as equivalent to a one-byte TXT record containing a single zero byte (i.e. a single empty string).

6.2 DNS TXT Record Format Rules for use in DNS-SD

DNS-SD uses DNS TXT records to store arbitrary name/value pairs conveying additional information about the named service. Each name/value pair is encoded as its own constituent string within the DNS TXT record, in the form "name=value". Everything up to the first '=' character is the name. Everything after the first '=' character to the end of the string (including subsequent '=' characters, if any) is the value. Specific rules governing names and values are given below. Each author defining a DNS-SD profile for discovering instances of a particular type of service should define the base set of name/value attributes that are valid for that type of service.

Using this standardized name/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 attribute names in a service TXT record, it MUST silently ignore them.

The TCP (or UDP) port number of the service, and target host name, are given in the SRV record. This information -- target host name and port number -- MUST NOT be duplicated using name/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 TCP-based application protocol, it should be possible, knowing only the host name and port number, to open a connection to that listening process, and then perform version- or feature-negotiation to determine the capabilities of the service instance. For example, when connecting to an AppleShare server over TCP, the client enters into a protocol exchange with the server to determine which version of the AppleShare protocol the server implements, and which optional features or capabilities (if any) are available. For a well-designed application protocol, clients should be able to connect and use the service even if there is no information at all in the TXT record. In this case, the 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. Extreme care should be taken when doing this to ensure that the information in the TXT record is in agreement with the information retrieved by a client connecting over TCP.

There are legacy protocols which 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 the old Unix LPR (port 515) protocol, the LPR service provides no way for the client to determine whether a particular printer accepts PostScript, or what version of PostScript, etc. In this case it is appropriate to embed this information in the TXT record, because the alternative is worse -- passing around written instructions to the users, arcane manual configuration of "/etc/printcap" files, etc.

6.3 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), keeping the total size under 400 bytes should allow it to fit in a single standard 512-byte DNS message. (This standard DNS message size is defined in RFC 1035.)

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.

6.4 Rules for Names in DNS-SD Name/Value Pairs

The "Name" MUST be at least one character. Strings beginning with an '=' character (i.e. the name is missing) SHOULD be silently ignored.

The characters of "Name" MUST be printable US-ASCII values (0x20-0x7E), excluding '=' (0x3D).

Spaces in the name 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 name, so "papersize=A4", "PAPERSIZE=A4" and "Papersize=A4" are all identical.

If there is no '=', then it is a boolean attribute, and is simply identified as being present, with no value.

A given attribute name may appear at most 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 names to values, and then make that abstraction available to client code. The rule that a given attribute name 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 attribute name 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 name/value pairs into a hash table, using the name as the hash table key, this means that if the implementation attempts to add a new name/value pair into the table and finds an entry with the same name already present, then the new entry being added should be silently discarded instead. For client implementations that retrieve name/value pairs by searching the TXT record for the requested name, they should search the TXT record from the start, and simply return the first matching name they find.

When examining a TXT record for a given named attribute, there are therefore four broad categories of results which may be returned:

* Attribute not present (Absent)

* Attribute present, with no value
  (e.g. "Anon Allowed" -- server allows anonymous connections)

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

* Attribute present, with non-empty value
  (e.g. "Installed 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:

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

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 name/value keys used by DNS-SD network printers.)

As a general rule, attribute names that contain no dots are defined as part of the open-standard definition written by the person or group defining the DNS-SD profile for discovering that particular service type. Vendor-specific extensions should be given names of the form "keyname.company.com=value", using a domain name legitimately registered to the person or organization creating the vendor-specific key. This reduces the risk of accidental conflict if different organizations each define their own vendor-specific keys.

6.5 Rules for Values in DNS-SD Name/Value Pairs

If there is an '=', then everything after the first '=' to the end of the string is the value. The value can contain any eight-bit values including '='. Leading or trailing spaces are part of the value, so don't put them there unless you intend them to be there. Any quotation marks around the value are part of the value, so don't put them there unless you intend them to be part of the value.

The value is opaque binary data. Often the value for a particular attribute will be US-ASCII (or UTF-8) text, but it is legal for a value to be any binary data. For example, if the value of a key is an IPv4 address, that address should simply be stored as four bytes of binary data, not as a variable-length 7-15 byte ASCII string giving the address represented in textual dotted decimal notation.

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 convert binary attribute data types into printable text (e.g. using hexadecimal, Base-64 or UU encoding) merely for the sake of making the data 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 name/value pairs. (The meaning of these name/value pairs, if any, would depend on the definitions pertaining to the service in question that is using them.)

```
---------------------------------------------------------------
| 0x0A | name=value | 0x08 | paper=A4 | 0x0E | DNS-SD Is Cool |
---------------------------------------------------------------
```

6.7 Version Tag

It is recommended that authors defining DNS-SD profiles include an attribute of the form "txtvers=xxx" in their definition, and require it to be the first name/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 specification provides useful insurance should incompatible changes become unavoidable. 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 TXT record specification being used to create this 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 name for this is "protovers".

7. Application Protocol Names

The <Service> portion of a Service Instance Name consists of a pair of DNS labels, following the established convention for SRV records [RFC 2782], namely: the first label of the pair is an underscore character followed by the Application Protocol Name, and the second label is either "_tcp" or "_udp".

Application Protocol Names may be no more than fourteen characters (not counting the mandatory underscore), conforming to normal DNS host name rules: Only lower-case letters, digits, and hyphens; must begin and end with lower-case letter or digit.

Wise selection of an Application Protocol Name is very important, and the choice is not always as obvious as it may appear.

In some cases, the Application Protocol 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 local network music playing protocol implemented by iTunes on Macintosh and Windows is little more than "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". Advertising "_http._tcp" service would cause iTunes servers to show up in conventional Web browsers (Safari, Camino, OmniWeb, Opera, Netscape, Internet Explorer, etc.) which is little use since it offers no pages containing human-readable content. Similarly, browsing for "_http._tcp" service would cause iTunes to find generic web servers, such as the embedded web servers in devices like printers, which is little use since printers generally don't have much music to offer.

Similarly, NFS is built on top of SUN RPC, but that doesn't mean it makes sense for an NFS server to advertise that it provides "SUN RPC" service. Likewise, Microsoft 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 usefully use the service. Merely advertising that a service was built on top of SUN RPC is no use if the client has no idea what the service actually does.

Another common mistake is to assume that the service type advertised by iTunes should be "_daap._http._tcp." This is also incorrect. Similarly, a protocol designer implementing a network service that happens to use Simple Object Access Protocol [SOAP] should not feel compelled to have "_soap" appear somewhere in the Application Protocol 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 structure of a service name has to reflect the internal structure of how the protocol was implemented. This is not correct. All that is required is that the service be identified by a unique Application Protocol Name. Making the Application Protocol Name at least marginally descriptive of what the service does is desirable, though not essential.

The "_tcp" or "_udp" should be regarded as little more than boilerplate text, and care should be taken not to attach too much importance to it. Some might argue that the "_tcp" or "_udp" should not be there at all, but this format is defined by RFC 2782, and that's not going to change. In addition, the presence of "_tcp" has the useful side-effect that it provides a convenient delegation point to hand off responsibility for service discovery to a different DNS server, if so desired.

7.1. Selective Instance Enumeration

This document does not attempt to define an arbitrary query language for service discovery, nor do we believe one is necessary.

However, there are some circumstances where narrowing the list of results may be useful. A hypothetical Web browser client that is able to retrieve HTML documents via HTTP and display them may also be able to retrieve HTML documents via FTP and display them, but only in the case of FTP servers that allow anonymous login. For that Web browser, discovering all FTP servers on the network is not useful. The Web browser only wants to discover FTP servers that it is able to talk to. In this case, a subtype of "_ftp._tcp" could be defined. Instead of issuing a query for "_ftp._tcp.<Domain>", the Web browser issues a query for "_anon._sub._ftp._tcp.<Domain>", where "_anon" is a defined subtype of "_ftp._tcp". The response to this query only includes the names of SRV records for FTP servers that are willing to allow anonymous login.

Note that the FTP server's Service Instance Name is unchanged -- it is still something of the form "The Server._ftp._tcp.example.com." The subdomain in which FTP server SRV records are registered defines the namespace within which FTP server names are unique. Additional subtypes (e.g. "_anon") of the basic service type (e.g. "_ftp._tcp") serve to narrow the list of results, not to create more namespace.

Subtypes are appropriate when it is desirable for different kinds of clients to be able to browse for services at two levels of granularity. In the example above, we hypothesize two classes of ftp client: clients that can provide username and password when connecting, and clients that can only do anonymous login. The set of ftp servers on the network is the same in both cases; the difference is that the more capable client wants to discover all of them, whereas the more limited client only wants to find the subset of those ftp servers that it can talk to. 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 an Application Protocol 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."

As with the TXT record name/value pairs, the list of possible subtypes, if any, are defined and specified separately for each basic service type. Note that the example given here using "_ftp" is a hypothetical one. The "_ftp" service type does not (currently) have any subtypes defined. Subtypes are currently a little-used feature of DNS-SD, and experience will show whether or not they ultimately prove to have broad applicability.

7.2 Service Name Length Limits

As described above, application protocol names are allowed to be up to fourteen characters long. The reason for this limit is to leave as many bytes of the domain name as possible available for use by both the network administrator (choosing service domain names) and the end user (choosing instance names).

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

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

The first example shows a service instance name, i.e. the name of the service's SRV and TXT records. The second shows a subtype browsing name, i.e. the name of a PTR record pointing to service instance names (see "Selective Instance Enumeration").

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.

The application protocol name <app> may be up to 14 bytes, plus the underscore and length byte, making 16. Including the "_udp" or "_tcp" and its length byte, this makes 21 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.

The terminating root label at the end counts as one byte.

Of our available 255 bytes, we have now accounted for 69+21+64+1 = 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 Application Protocol Names to 14 characters or less. Allowing more characters would not add to the expressive power of the protocol, and would needlessly lower the limit on the maximum <parentdomain> length that may be safely used.

8. Flagship Naming

In some cases, there may be several network protocols available which all perform roughly the same logical function. For example, the printing world has the LPR protocol, and the Internet Printing Protocol (IPP), 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 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 the case of multiple protocols like this that all perform effectively the same function, the client should suppress duplicate names and display each name only once. When the user prints to a given named printer, the printing client is responsible for choosing the protocol which 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 some future printer that only supports IPP printing, and some other future printer that only supports pdl-datastream printing. The name spaces 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 have two different printers both called "Sales Department", just because those printers are implementing different protocols.

To help guard against this, when there are two or more network protocols which 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 = hostname 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 entity implementing at least one of the protocols from the class, 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 class will detect a conflict when they try to use it. The SRV record needs to contain the target host name in order for the 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 so no conflict would be registered.

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.

9. Service Type Enumeration

In general, clients are not interested in finding *every* service on the network, just the services that the client knows how to talk to. (Software designers may *think* there's some value to finding *every* service on the network, but that's just wooly thinking.)

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 reason, a special meta-query is defined. A DNS query for PTR records with the name "_services._dns-sd._udp.<Domain>" yields a list of PTR records, where the rdata of each PTR record is the name of a service type. A subsequent query for PTR records with one of those names yields a list of instances of that service type.

10. Populating the DNS with Information

How the SRV and PTR records that describe services and allow them to be enumerated make their way into the DNS is outside the scope of this document. However, it can happen easily in any of a number of ways, for example:

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 Apple LaserWriters 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 DNS-SD clients that don't implement AppleTalk NBP, and don't want to.

Future IP printers could use Dynamic DNS Update [RFC 2136] to automatically register their own SRV and PTR records with the DNS server.

A printer manager device which has knowledge of printers on the network through some other management protocol could also use Dynamic DNS Update [RFC 2136].

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.

Zeroconf printers answer Multicast DNS queries on the local link for appropriate PTR and SRV names ending with ".local." [mDNS]

11. Relationship to Multicast DNS

DNS-Based Service Discovery is only peripherally related to Multicast DNS, in that the standard unicast DNS queries used by DNS-SD may also be performed using multicast when appropriate, which is particularly beneficial in Zeroconf environments [ZC].

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

One of the main reasons for DNS-Based Service Discovery is so that when a visiting client (e.g. a laptop computer) arrives at a new network, it can discover what services are available on that network without manual configuration. This logic that applies to discovering services without manual configuration also applies to discovering the domains in which services are registered without requiring manual configuration.

This discovery is performed recursively, 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 final query shown yields the "legacy browsing" domain. Sophisticated client applications that care to present choices of domain to the user, use the answers learned from the previous four queries to discover those 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 "legacy browsing" query is provided, to allow the network administrator to communicate to the client operating systems which domain should be used for these applications.

These domains are purely advisory. The client or user is free to browse and/or register services 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 they may make a suitable selection, or ignore the offered suggestions and manually enter their own choice.

The <domain> part of the 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) [RFC 2132] or the DHCP "Domain Search" option (option code 119) [RFC 3397].

The <domain> part of the name may also be derived 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. 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 address 192.168.12.34, with subnet mask 255.255.0.0, then the 'base' address of the subnet is 192.168.0.0, and to discover the recommended legacy browsing domain 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."

Sophisticated clients may perform domain enumeration queries both in "local" and in one or more unicast domains, and then present the user with an aggregate result, combining the information received from all sources.

13. 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 typically records that the client did not explicitly request, but the server has reasonable grounds to expect that the client might request them shortly.

This section recommends which additional records should be generated to improve network efficiency for both unicast and multicast DNS-SD responses.

13.1 PTR Records

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

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

13.2 SRV Records

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

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

13.3 TXT Records

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

13.4 Other Record Types

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

14. Comparison with Alternative Service Discovery Protocols

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. The Zeroconf host profile [ZCHP] requires the use of a DNS-like protocol over IP Multicast for host name resolution in the absence of DNS servers. Given that Zeroconf hosts will have to implement this Multicast-based DNS-like protocol anyway, it makes sense for them to also perform service discovery using that same Multicast-based DNS-like software, instead of also having to implement an entirely different service discovery protocol.

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 [RFC 2136]. Given that virtually every company already has to operate and maintain a DNS server anyway, it makes sense to take advantage of this 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>" subdomain 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 that is running, 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 [RFC 2535].

In summary:

Service discovery requires a central aggregation server.
DNS already has one: It's called a DNS server.

Service discovery requires a service registration protocol.
DNS already has one: It's called DNS Dynamic Update.

Service discovery requires a query protocol
DNS already has one: It's called DNS.

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

Service discovery requires a multicast mode for ad-hoc networks.
Zeroconf environments already require a multicast-based DNS-like name lookup protocol for mapping host names to addresses, so it makes sense to let one multicast-based protocol do both jobs.

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.

15. Real 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, but if you wish you can try these examples for yourself as you read along, using the command-line tools already available on your own Unix machine.

15.1 Question: What FTP servers are being advertised from dns-sd.org?

```
nslookup -q=ptr _ftp._tcp.dns-sd.org.
_ftp._tcp.dns-sd.org
         name = Apple\032QuickTime\032Files._ftp._tcp.dns-sd.org
_ftp._tcp.dns-sd.org
         name = Microsoft\032Developer\032Files._ftp._tcp.dns-sd.org
_ftp._tcp.dns-sd.org
         name = Registered\032Users'\032Only._ftp._tcp.dns-sd.org
```

Answer: There are three, called "Apple QuickTime Files", "Microsoft Developer Files" and "Registered Users' Only".

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

15.2 Question: What FTP servers allow anonymous access?

```
nslookup -q=ptr _anon._sub._ftp._tcp.dns-sd.org
_anon._sub._ftp._tcp.dns-sd.org
         name = Apple\032QuickTime\032Files._ftp._tcp.dns-sd.org
_anon._sub._ftp._tcp.dns-sd.org
         name = Microsoft\032Developer\032Files._ftp._tcp.dns-sd.org
```

Answer: Only "Apple QuickTime Files" and "Microsoft Developer Files" allow anonymous access.

15.3 Question: How do I access "Apple QuickTime Files"?

```
nslookup -q=any "Apple\032QuickTime\032Files._ftp._tcp.dns-sd.org."
Apple\032QuickTime\032Files._ftp._tcp.dns-sd.org
          text = "path=/quicktime"
Apple\032QuickTime\032Files._ftp._tcp.dns-sd.org
          priority = 0, weight = 0, port= 21 host = ftp.apple.com
ftp.apple.com   internet address = 17.254.0.27
ftp.apple.com   internet address = 17.254.0.31
ftp.apple.com   internet address = 17.254.0.26
```

Answer: You need to connect to ftp.apple.com, port 21, path "/quicktime". The addresses for ftp.apple.com are also given.

16. User Interface Considerations

DNS-Based Service Discovery was designed by first giving careful consideration to what constitutes a good user experience for service discovery, and then designing a protocol with the features necessary to enable that good user experience. This section covers two issues in particular: Choice of factory-default names (and automatic renaming behavior) for devices advertising services, and the "continuous live update" user-experience model for clients browsing to discover services.

16.1 Service Advertising User-Interface Considerations

When a DNS-SD service is advertised using Multicast DNS [mDNS], automatic name conflict and resolution will occur if there is already another service of the same type advertising with the same name. As described in the Multicast DNS specification [mDNS], upon 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 the key to providing user-friendly service names in the out-of-the-box factory-default configuration. Some product developers may not have realized this, because there are some products today where the factory-default name is distinctly unfriendly, containing random-looking strings of characters, like 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 useful to human users. If the name is not unique on the local network the protocol will rememdy this as necessary. It is ironic that many of the devices with this 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). Examples of good factory-default names are as follows:

```
Brother 5070N
Canon W2200                             [ Apologies to makers of ]
HP LaserJet 4600                        [ DNS-SD/mDNS printers   ]
Lexmark W840                            [ not listed. Email      ]
Okidata C5300                           [ the authors and we'll  ]
Ricoh Aficio CL7100                     [ add you to the list.   ]
Xerox Phaser 6200DX
```

To complete the case for why adding long ugly 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 it detect that the name "Printer" is already in use and automatically instead name itself "Printer (2)" provides a good user experience. For the users, remembering that the old printer is "Printer" and the new one is "Printer (2)" is easy and intuitive. Seeing two printers called "Printer 0001E68C74FB" and "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 desirable 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.

16.2 Client Browsing User-Interface Considerations

Of particular concern in the design of DNS-SD was the dynamic nature of service discovery in a changing network environment. Other service discovery protocols have been designed with an implicit unstated assumption that the usage model is:

(a) client calls the service discovery code
(b) client gets list of discovered services
    as of a particular instant in time, and then
(c) client displays list for 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 user successfully finds the service they're looking for. In the case where the user is looking for (say) a particular printer, and that printer's 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 (or, worse, dismiss the browsing window entirely, and open a new one to initiate a new network search attempt) 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 typically takes a few seconds. A fairly typical user experience model 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, the moment it's displayed on the screen it's already beginning to get stale and out-of-date.

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

1. Displaying a list of discovered services should be effectively instantaneous -- i.e. typically 1/10 second, 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 initial stable list has been built and displayed, the list 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 to be in the habit of leaving service discovery windows open, and coming to expect to be able to rely on them to show a continuous 'live' view of current network reality,

this creates a new requirement for us: 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 result would be unacceptable: the service discovery list would simply grow monotonically over time, and would require a periodic "refresh" (or complete dismissal and recreation) to clear out old stale data.

4. With users getting to be in the habit of leaving service discovery windows open, these windows need to 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 no Ethernet cable is connected to the client machine, and the window appears empty with no discovered services, then when the user connects the cable the window should automatically populate with discovered services without requiring any explicit user action. If the user disconnects the Ethernet cable, all the services discovered via that network interface should automatically disappear. If the user switches from one 802.11 wireless base station to another, the service discovery window should automatically update to remove all the services discovered via the old wireless base station, and add all the services discovered via the new one.

If these requirements seem to be setting an arbitrary and unreasonably high standard for service discovery, bear in mind that while it may have seemed that way to some, back in the 1990s when these ideas were first proposed, in the years since then Apple and other companies have shipped multiple implementations of DNS-SD/mDNS that meet and exceed these requirements. In the years since Apple shipped Mac OS X 10.2 Jaguar with the Open Source mDNSResponder daemon, this service discovery "live browsing" paradigm has been adopted and implemented in a wide range of Apple and third-party applications, including printer discovery, Safari discovery of devices with embedded web servers (for status and configuration), iTunes music sharing, iPhoto photo sharing, the iChat Bonjour buddy list, SubEthaEdit multi-user document editing, etc.

With so many different applications demonstrating that the "live browsing" paradigm is clearly achievable, these four requirements should not be regarded as idealistic unattainable goals, but instead as the bare minimum baseline functionality that any credible service discovery protocol needs to achieve.

17. IPv6 Considerations

IPv6 has no significant differences, except that the address of the SRV record's target host is given by the appropriate IPv6 address records instead of the IPv4 "A" record.

18. Security Considerations

DNSSEC [RFC 2535] should be used where the authenticity of information is important. Since DNS-SD is just a naming and usage convention for 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.

19. IANA Considerations

This protocol builds on DNS SRV records [RFC 2782], and similarly requires IANA to assign unique application protocol names. Unfortunately, the "IANA Considerations" section of RFC 2782 says simply, "The IANA has assigned RR type value 33 to the SRV RR. No other IANA services are required by this document." Due to this oversight, IANA is currently prevented from carrying out the necessary function of assigning these unique identifiers.

This document proposes the following IANA allocation policy for unique application protocol names:

```
Allowable names:
  * Must be no more than fourteen characters long
  * Must consist only of:
    - lower-case letters 'a' - 'z'
    - digits '0' - '9'
    - the hyphen character '-'
  * Must begin and end with a lower-case letter or digit.
  * Must not already be assigned to some other protocol in the
    existing IANA "list of assigned application protocol names
    and port numbers" [ports].
```

These identifiers are allocated on a First Come First Served basis. In the event of abuse (e.g. automated mass registrations, etc.), the policy may be changed without notice to Expert Review [RFC 2434].

The textual nature of service/protocol names means that there are almost infinitely many more of them available than the finite set of 65535 possible port numbers. This means that developers can produce experimental implementations using unregistered service names with little chance of accidental collision, providing service names are chosen with appropriate care. However, this document strongly advocates that on or before the date a product ships, developers should properly register their service names.

Some developers have expressed concern that publicly registering their service names (and port numbers today) with IANA before a product ships may give away clues about that product to competitors. For this reason, IANA should consider allowing service name applications to remain secret for some period of time, much as US patent applications remain secret for two years after the date of filing.

This proposed IANA allocation policy is not in force until this document is published as an RFC. In the meantime, unique application protocol names may be registered according to the instructions at <http://www.dns-sd.org/ServiceTypes.html>. As of August 2006, there are roughly 300 application protocols in currently shipping products that have been so registered as using DNS-SD for service discovery.

20. Acknowledgments

The concepts described in this document have been explored, developed and implemented with help from Richard Brown, Erik Guttman, Paul Vixie, and Bill Woodcock.

Special thanks go to Bob Bradley, Josh Graessley, Scott Herscher, Roger Pantos and Kiren Sekar for their significant contributions.

21. Deployment History

The first implementations of DNS-Based Service Discovery and Multicast DNS were initially developed during the late 1990s, but the event that put them into the media spotlight was Steve Jobs demonstrating it live on stage in his keynote presentation opening Apple's annual Worldwide Developers Conference in May 2002, and announcing Apple's adoption of the technology throughout its hardware and software product line. Three months later, in August 2002, Apple shipped Mac OS X 10.2 Jaguar, and millions of end-users got their first exposure to Zero Configuration Networking with DNS-SD/mDNS in applications like Safari, iChat, and printer setup. A month later, in September 2002, Apple released the entire source code for the mDNS Responder daemon under its Darwin Open Source project, with code not just for Mac OS X, but also for a range of other platforms including Windows, VxWorks, Linux, Solaris, FreeBSD, etc.

Many hardware makers were quick to see the benefits of Zero Configuration Networking. Printer makers especially were enthusiastic early adopters, and within a year every major printer manufacturer was shipping DNS-SD/mDNS-enabled network printers. If you've bought any network printer at all in the last few years, it was probably one that supports DNS-SD/mDNS, even if you didn't know that at the time. For Mac OS X users, telling if you have DNS-SD/mDNS printers on your network is easy because they automatically appear in the "Bonjour" submenu in the "Print" dialog of every Mac application. Microsoft Windows users can get a similar experience by installing Bonjour for Windows (takes about 90 seconds, no restart required) and running the Bonjour for Windows Printer Setup Wizard [B4W].

The Open Source community has produced several independent implementations of DNS-Based Service Discovery and 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.

22. Copyright Notice

23. Normative References

[ports]   IANA list of assigned application protocol names and port numbers <http://www.iana.org/assignments/port-numbers>

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

[RFC 1034] Mockapetris, P., "Domain Names - Concepts and Facilities", STD 13, RFC 1034, November 1987.

[RFC 1035] Mockapetris, P., "Domain Names - Implementation and Specifications", STD 13, RFC 1035, November 1987.

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

[RFC 2782] Gulbrandsen, A., et al., "A DNS RR for specifying the location of services (DNS SRV)", RFC 2782, February 2000.

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

[UAX15]   "Unicode Normalization Forms" http://www.unicode.org/reports/tr15/

24. Informative References

[B4W]   Bonjour for Windows <http://www.apple.com/bonjour/>

[mDNS]   Cheshire, S., and M. Krochmal, "Multicast DNS", Internet-Draft (work in progress), draft-cheshire-dnsext-multicastdns-06.txt, August 2006.

[NBP]   Cheshire, S., and M. Krochmal, "Requirements for a Protocol to Replace AppleTalk NBP", Internet-Draft (work in progress), draft-cheshire-dnsext-nbp-05.txt, August 2006.

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

[RFC 2136] Vixie, P., et al., "Dynamic Updates in the Domain Name System (DNS UPDATE)", RFC 2136, April 1997.

[RFC 2434] Narten, T., and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 2434, October 1998.

[RFC 2535] Eastlake, D., "Domain Name System Security Extensions", RFC 2535, March 1999.

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

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

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

[ZC] Williams, A., "Requirements for Automatic Configuration of IP Hosts", Internet-Draft (work in progress), draft-ietf-zeroconf-reqts-12.txt, September 2002.

[ZCHP] Guttman, E., "Zeroconf Host Profile Applicability Statement", Internet-Draft (work in progress), draft-ietf-zeroconf-host-prof-01.txt, July 2001.

25. Authors' Addresses

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

Phone: +1 408 974 3207
EMail: rfc [at] stuartcheshire [dot] org

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

Phone: +1 408 974 4368
EMail: marc [at] apple [dot] com