DNSServiceDiscovery Mach-Based API

(Not Recommended)

Networking > Bonjour

2005-04-29

Apple Inc.
© 2001, 2005 Apple Computer, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, AppleTalk, Bonjour, Carbon, Cocoa, eMac, Mac, Mac OS, and Macintosh are trademarks of Apple Inc., registered in the United States and other countries.

Numbers is a trademark of Apple Inc.

UNIX is a registered trademark of The Open Group

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Figures and Listings

Legacy Document | 2005-04-29 | © 2001, 2005 Apple Computer, Inc. All Rights Reserved.

Introduction

Framework:	None.
Declared in	DNSServiceDiscovery.h

Important: The information in this document is obsolete and should not be used for new development.

The DNSServiceDiscovery API is part of Bonjour, Apple's implementation of zero-configuration networking (ZEROCONF).

Bonjour allows you to register a network service, such as a printer or file server, so that it can be found by name or browsed for by service type and domain. Using Bonjour, applications can discover what services are available on the network, along with all necessary access information—such as name, IP address, and port number—for a given service.

In effect, Bonjour combines the functions of a local DNS server and AppleTalk. Bonjour allows applications to provide user-friendly printer and server browsing, among other things, over standard IP networks. It does this by combining protocols such as multicast and DNS to add new functionality to the network (such as multicast DNS).

This gives applications easy access to services over local IP networks without requiring the service or the application to support an AppleTalk or Netbeui stack, and without requiring a DNS server for the local network.

For a detailed overview of Bonjour, see "Bonjour Overview" (page 9)

For additional information on Bonjour, including links to standards, specifications, and resources, see http://developer.apple.com/networking/bonjour/.

This section describes the functions, callbacks, and data structures that make up the DNSServiceDiscovery API.

Who Should Use this API

The DNSServiceDiscovery API is appropriate for Darwin programmers and developers who are comfortable working in Mac OS X at the Mach level. This is a low-level API that interacts directly with the Mach kernel and the mDNS responder daemon.

Bonjour provides two higher-level APIs as alternatives to this one. Cocoa developers should generally use the NSNetServices API documented in *Bonjour Overview* and Carbon developers should generally use the CFNetServices API documented in *CFNetwork Programming Guide*.

These higher-level APIs provide the similar services to the DNSServiceDiscovery API and are designed to fit cleanly with code written using a specific framework. Note, however, that the DNSServiceDiscovery API provides a finer degree of control than the CFNetServices and NSNetServices APIs.

Requirements

Version 10.2 or later of the Mac OS X or Darwin operating system is required.

This API requires the services of an mDNS responder. Mac OS X, versions 10.2 and later, include an mDNS responder daemon as part of the operating system. Apple also provides source code for an mDNS responder to Darwin developers as part of versions 10.2 and later.

Note: Apple encourages hardware developers to embed the Darwin mDNS responder code in their hardware.

Limitations

The DNSServiceDiscovery API does not perform network access setup for services. Similarly, the DNSServiceDiscovery API does not provide a network connection from an application to a service. It allows applications to browse for services, or to ask for them by name, and provides the IP address, port, and so on. Use the NSNetwork, CFNetwork, or BSD sockets API to actually connect to a service over the network.

Bonjour Overview

Bonjour is a powerful new system for finding and publishing network services, such as Web servers, file servers, and printers, on Internet Protocol (IP) networks. This system adds the simplicity and ease-of-use of AppleTalk to the power of IP network services.

Bonjour is Apple's implementation of zero-configuration networking over IP and is based on IETF standard protocols (IP, ARP, DNS, etc.). Bonjour comes out of the work of the IERF Zeroconf Working Group.

This section describes some of the problems that Bonjour solves and how it solves them.

Why Bonjour?

Over the last twenty years, a large number of wide-area networking protocols have appeared and disappeared. In recent years, Internet protocol (IP) has become the single predominant standard on every computing platform. The majority of computers, and many other network devices, all speak a common language. For wide-area networks and the Internet, IP protocol is all you need.

On local area networks (LANs) however, a few other protocols live on. To work on as many local networks as possible, printers and applications such as multiplayer games must support not only IP, but also AppleTalk or Windows Netbeui, or perhaps all three. While IP has emerged as a unifying protocol for wide area networks and the Internet, it is not a universal standard on local networks, especially small networks and home networks. Why?

The answer is simple: these networks don't often have dedicated address and name servers (DHCP and DNS) or a real system administrator.

For IP to work, every device needs a unique address. To make this happen, either everyone must agree on static IP addresses and manually type them in, or someone must set up a DHCP server or similar service to dynamically allocate addresses to clients. To refer to services by name, someone must set up a DNS server to perform the name-to-address translation, and, typically, use "well known ports" for specific types of services. To use a service, network users have to know it's name, so when a service is added, everyone needs to be notified. Someone with a lot of knowledge has to set all this up and maintain it.

More and more, people that do not fit the traditional role of the network administrator are setting up networks. Families are setting up home networks so they can share printers, files, and Internet connections. Peers meeting at conferences are setting up ad-hoc networks to exchange data. Even inside well-managed corporate networks, employees are adding devices to and removing devices from their local subnets. Currently, all these activities require manual configuration of IP addresses and names.

This new breed of network administrator does not want to configure subnet masks or DNS servers. Even a highly competent network administrator doesn't want to send email to every employee every time a printer is added to the network. Printer manufacturers and game publishers don't want to support multiple protocol stacks on a $50 product. People need to be able to plug in a printer, or plug two laptops together, or look for a file server or game server on the local network, without wasting time trying to get the configuration right.

Once the configuration of devices on an IP network is right, the user needs to know the exact name of any printer or other service in order to use it. That's better than typing in an IP address, but it doesn't help the user find services he or she doesn't already know about. And it doesn't forgive spelling errors. Browsing for available services is often simply impossible. A large number of IP service browsing protocols have appeared and disappeared, but none has achieved critical mass.

Before the emergence of IP as the preeminent interoperative networking protocol, AppleTalk solved the configuration and usage problems that continue to hinder IP today. With AppleTalk, users can simply browse for a service and click to choose it. For example, if you connect a group of Macintosh computers running Mac OS 9 or earlier with an Ethernet hub, they can instantly see all the available printers, file servers, and other services available on the local network. All this happens without centralized allocation of network addresses, without a centralized name server, and without a centralized repository of available services.

People need a simple and reliable way to configure and browse for services over IP networks. They want to discover available services and choose one from a list, instead of having to know each service's name or IP address in advance. It is in everyone's interest for IP to have this capability. This is exactly the capability that Bonjour provides.

Zero Configuration: An Example

The potential for zero-configuration IP networking is tremendous. Consider the everyday task of printing. Once a printer is configured on your computer, it's simply a matter of choosing an application's Print command.

Take your laptop to a client's company, or a neighbor's house, and try to print something. If they have a printer that supports Bonjour protocols, printing is just as easy as it was on your local network. To print, connect an Ethernet cable from your laptop to your client's LAN and start up your laptop. Or start up your laptop and it instantly finds your neighbor's home wireless network. Either way, your laptop automatically discovers any available printers. You open the document, choose the Print command, and every available printer appears in the Print dialog. You select a printer, click Print, and the document prints.

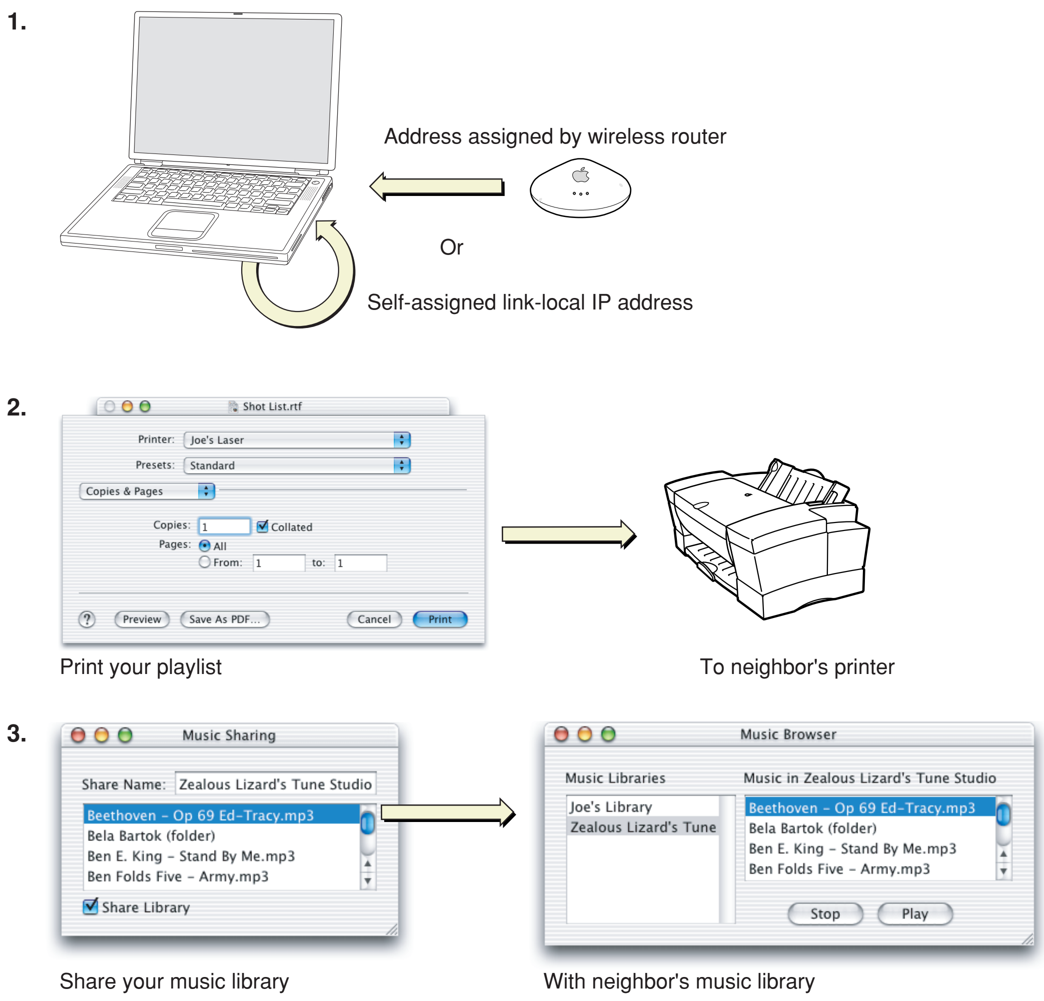
Or say you want to play a network game with a friend. You open the game, and your friend's copy of the game instantly sees your copy over the network. Or if you have a music sharing application on both computers, the programs themselves can discover each other and instantly swap songlists. Similarly, if you have a shared folder or have personal Web sharing turned on, your shared files and Web pages are instantly available to others.

This scenario is illustrated in Figure 1-1 (page 11). In step 1, you open up your laptop in your neighbor's house, and the laptop either obtains an address from the DHCP server in the router or, in the absence of a DHCP server, assigns itself an available local address. In step 2, the network is queried for available printers so that when you open the Print dialog, your neighbor's printer is listed. Finally, in step 3, you turn on music sharing on your computer, and your neighbor's computer sees it and connects.

These are just a few of the existing applications that can benefit from zero-configuration IP networking. Zero-configuration IP networking has the potential to enhance contact management, PDA synchronization, distributed processing, and many other network applications. Additionally, zero-configuration IP networking opens the door for a whole new class of IP-enabled digital devices.

Legacy Document | 2005-04-29 | © 2001, 2005 Apple Computer, Inc. All Rights Reserved.

Figure 1-1 A typical zero-configuration networking session



Shot List.rtf

Printer: Joe's Laser

Presets: Standard

Copies & Pages

Copies: 1 [✓ Collated]

Pages: [● All] ○ From: 1 to: 1

Music Sharing

Share Name: Zealous Lizard's Tune Studio

Beethoven - Op 69 Ed-Tracy.mp3
Bela Bartok (folder)
Ben E. King - Stand By Me.mp3
Ben Folds Five - Army.mp3

[✓ Share Library]

Music Browser

Music Libraries	Music in Zealous Lizard's Tune Studio
Joe's Library	Beethoven - Op 69 Ed-Tracy.mp3
Zealous Lizard's Tune	Bela Bartok (folder)
	Ben E. King - Stand By Me.mp3
	Ben Folds Five - Army.mp3

What is Bonjour?

Bonjour is Apple's proposal for zero-configuration networking over IP. Bonjour comes out of the work of the ZEROCONF Working Group, part of the Internet Engineering Task Force (IETF). The ZEROCONF Working Group's requirements and proposed solutions essentially cover three areas:

- addressing (allocating IP addresses to hosts)
- naming (using names to refer to hosts instead of IP addresses)
- service discovery (finding services on the network automatically)

Bonjour has a zero-configuration solution for all three of these areas, as described in the following three sections.

Bonjour allows service providers, hardware manufacturers, and application programmers to support a single network protocol—IP—while breaking new ground in ease of use.

Legacy Document | 2005-04-29 | © 2001, 2005 Apple Computer, Inc. All Rights Reserved.

Network users no longer have to assign IP addresses, assign host names, or even type in names to access services on the network. Users simply ask to see what network services are available, and choose from the list.

In many ways, this kind of browsing is even more powerful for applications than for users. Applications can automatically detect services they need or other applications they can interact with, allowing automatic connection, communication, and data exchange, without requiring user intervention.

Addressing

The addressing problem is solved by self-assigned link-local addressing. Link-local addressing uses a range of addresses reserved for the local network, typically a small LAN or a single LAN segment.

Self-assigned addressing is simply picking a random IP address in the link-local range and testing it. If the address in not use, it is now your local address. If it is already in use, pick another address and try again.

Note: Two hosts are considered to be on the same local link if, when one host sends packets to the other, the entire link-layer payload (the content of the packet as represented in the physical network, such as Ethernet) arrives unmodified. In practice, on an Ethernet network, this means that no IP router touches the packet between the two hosts.

Self-assigned link-local addressing has already shipped on IPv4 starting with Mac OS 8.5, Windows 98, and Mac OS X version 10.0. The IPv6 specification includes self-assigned link-local addressing.

Any user or service on a computer that supports self-assigned link-local addressing benefits from this feature automatically. When your host computer encounters a local network, it finds an unused local address and adopts it. No action on your part is required.

Hardware manufacturers should implement self-assigned link-local addressing on their devices to obtain the full benefit of Bonjour.

Naming

The proposed solution for name-to-address translation on a local network uses multicast DNS (mDNS), in which DNS-format queries are sent over the local network using IP multicast. Because these DNS queries are sent to a multicast address, no single DNS server with global knowledge is required to answer the queries. Each service or device can provide its own DNS capability—when it sees a query for its own name, it provides a DNS response with its own address.

Bonjour goes a bit further. It includes a responder that handles mDNS queries for any network service on the host computer. This relieves your application of the need to interpret and respond to mDNS messages. Just register your service with the Bonjour mDNS responder, and any queries for your name will be directed to your address automatically.

Note: Registration is performed using one of the Bonjour APIs. This is available only to services running on the host computer. Services running on other devices, such as printers, need to implement a simple mDNS responder that handles queries for services provided by that device.

For name-to-address translation to work properly, you need a unique name on the local network. Unlike conventional DNS host names, the local name only has significance on the local network or LAN segment. You can self-assign a local name the same way you self-assign a local address—choose one; if it's not already in use, it's yours:

- Hardware manufacturers determine whether their chosen name is already use by sending an mDNS query for that name and looking for any response. If there is a response, choose another name. Devices without a user interface append an incrementally larger number to a default name until the name is unique. For example, if a printer with the default name `XYZ-LaserPrinter` attaches to a local network with two other identical printers already installed, it tests for `XYZ-LaserPrinter`, then `XYZ-LaserPrinter2`, then `XYZ-LaserPrinter3`, which is unused and becomes its name.
- Software services provide a local name when they register with Bonjour. If the provided name is already in use, Bonjour returns an error, and the service chooses another name.

Starting with Mac OS X version 10.2, users can set a local host name for their computers—the Local Hostname setting in the Sharing pane of System Preferences. The host name can be used anywhere a conventional DNS host name is used—Web browsers, command line tools, and so on. To indicate to the system that the name is a local host name, append a dot (`.`) and `local.` to the host name — `Steve.local.` is an example of a local host name.

Important: The first dot acts as a separator. The dot at the end of `local.` is required to prevent applications from searching for local services outside the local network.

For example, if a user types `steve.local.` into a Web browser, this tells the system to multicast the request for `steve` on the local network instead of sending it to the conventional DNS server. If a Bonjour-enabled computer named `steve` is on the local network, the user's browser is sent the correct IP address for it. This allows users to access local hosts and services without a conventional DNS server.

Note: Users can avoid typing `.local.` after Bonjour host names by entering `local` in the Search Domains section of the Network pane in System Preferences, along with any other DNS domains such as `apple.com` or `earthlink.net`. An unqualified name, such as `steve`, is searched for in successive domains listed in the Search Domains section of the Network pane, in this case `steve.apple.com`, `steve.earthlink.net`, and `steve.local`.

For more information, see "Bonjour and Domain Names" (page 16).

Service Discovery

The final element of Bonjour is service discovery. Service discovery allows applications to find all available instances of a particular type of service and to maintain a list of named services. The application can then resolve a named instance of a service to an IP address and port number, as described in "Naming" (page 12).

The list of named services provides a layer of indirection between a service and its current IP address. Indirection enables applications to keep a persistent list of available services and resolve an actual network address just prior to actually using a service. The list allows services to be relocated dynamically without generating a lot of network traffic announcing the change.

Service discovery in Bonjour is accomplished by "browsing." An mDNS query is sent out for a given service type and domain, and any matching servers reply with their names. The result is a list of available services to choose from.

This is very different from the traditional device-centric idea of network services. For someone who deals with servers, network devices, and network programming, it is easy to get in the habit of thinking about services in terms of physical hardware. In this device-centric view, the network consists of a number of devices or hosts, each with a set of services. For example, the network might consist of a server machine and several client machines. In a device-centric browsing scheme, a client queries the server for what services it is running, gets back a list (FTP, HTTP, and so on), and decides which service to use. The interface reflects the way the physical system is organized. But this is not necessarily what the user logically wants or needs.

Users typically want to accomplish a certain task, not query a list of devices to find out what services are running. It makes far more sense for a client to ask a single question: "What print services are available?" than to query each available device with the question, "What services are you running?" and sift through the results looking for printers. The device-centric approach is not only time-consuming, it generates a tremendous amount of network traffic, most of it useless. The service-centric approach sends a single query, generating only relevant replies.

Additionally, services are not tied to specific IP addresses or even host names. For example, a website may be hosted by multiple servers with different addresses. Within an organization, network administrators may need to move a service from one server to another to help balance the load. If clients store the host name (as in most cases they now do), they will not be able to connect if the service moves to a different host.

Bonjour takes the service-oriented view. Queries are made according to the type of service needed, not the hosts providing them. Applications store service names, not addresses, so if the IP address, port number, or even host name has changed, the application can still connect. By concentrating on services rather than devices, the user's browsing experience is made more useful and trouble-free.

Avoiding "Chattiness"

Server-free addressing, naming, and service discovery have the potential to create a significant amount of excess network traffic, but Bonjour takes a number of steps to reduce this traffic to a minimum. This allows Bonjour to attain AppleTalk's ease of use while avoiding any unnecessary "chattiness."

Bonjour makes use of several mechanisms for reducing zero-configuration overhead, including caching, suppression of duplicate responses, exponential back-off, and service announcement, as described in the following sections.

Caching

Bonjour uses a cache of multicast packets to prevent hosts from requesting information that has already been requested. For example, when one host requests, say, a list of LPR print spoolers, the list of printers comes back via multicast, so all local hosts see it. The next time a host needs a list of print spoolers, it already has the list in its cache and does not need to reissue the query. The multicast DNS responder is responsible for maintaining the cache; application developers do not need to do anything to maintain it.

Suppression of Duplicate Responses

To prevent repeated answers to the same query, Bonjour service queries include a list of known answers. For example, if a host is browsing for printers, the first query includes no print services and gets, say, twelve replies from available print servers. The next time the host queries for print services, the query includes a list of known servers. Print servers already on the list do not respond.

Bonjour suppresses duplicate responses in another way. If a host is about to respond, and notices that another host has already responded with the same information, the host suppresses its response.

Application developers do not need to take any action to suppress duplicate responses. Bonjour takes care of duplicate response suppression.

Exponential Back-off and Service Announcement

When a host is browsing for services, it does not continually send queries to see if new services are available. Instead, the host issues an initial query and sends subsequent queries exponentially less often: after 1 second, 2 seconds, 4 seconds, 8 seconds, and so on, up to a maximum interval of one hour.

This does not mean that it can take over an hour for a browser to see a new service. When a service starts up on the network, it announces its presence with the same exponential back-off algorithm. This way, network traffic for service announcement and discovery is kept to a minimum, but new services are seen very quickly.

Services running on a Bonjour-equipped host are announced automatically when they register with the mDNS responder. Services running on other hardware, such as printers, should implement service announcement with exponential back-off to take full advantage of Bonjour.

Programming in Bonjour

Bonjour provides API support that enables applications to register services they offer, browse for services on the network, and obtain the current IP address and port of a given service instance. Bonjour takes care of the low-level tasks behind these operations, such as announcing registered services, sending mDNS queries, tracking responses, and providing name-to-address translation for named services.

Four APIs are available:

- NSNetServices API, a Cocoa framework
- CFNetServices API, a Carbon framework
- Mach-based DNSServiceDiscovery API (described in this document and deprecated in favor of the socket-based DNSServiceDiscovery API)
- Socket-based DNSServiceDiscovery API

Cocoa programmers (Objective C) should use the NSNetServices API. However, Cocoa programmers who need to exercise finer control over events than offered by the NSNetServices API may want to use the DNSServiceDiscovery API.

Carbon programmers (C/C++) should use the CFNetServices API. Carbon programmers who are comfortable working with Mac OS X at the Mach level may find the Mach-based DNSServiceDiscovery API simpler to use for some applications. Note, however, that the Mach-based DNSServiceDiscovery API is deprecated, and developers are encouraged to use the new socket-based API.

Darwin programmers should use the socket-based DNSServiceDiscovery API.

Documentation on all four Bonjour APIs can be found in Bonjour Documentation on the ADC website.

Sample code, links to specifications, and other resources are available at http://developer.apple.com/networking/bonjour/.

Hardware developers should examine the sample code for an mDNS responder provided at http://developer.apple.com/darwin.

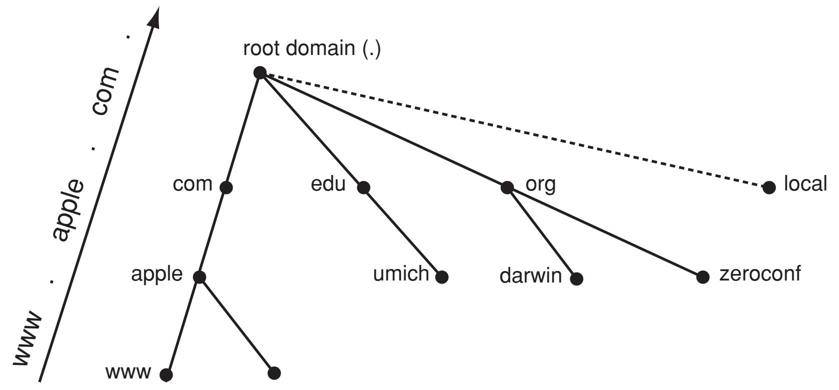
Bonjour and Domain Names

Bonjour names for service instances and service types are related to Domain Name System (DNS) domain names. This section explains DNS domain names, the Bonjour local "domain," and the naming rules for Bonjour service instances and service types.

Domain Names and DNS

DNS uses a specific-to-general naming scheme for domain names. The most general domain is `.` ("dot"), called the **root domain**, which is akin to the root directory `/` in a UNIX file system. Every other domain falls in a hierarchy below the root domain. For example, the name `www.apple.com.` is within the **second-level domain** `apple.com.`, which is within the **top-level domain** `com.`, which in turn is part of `.` ("dot"), the root domain. Figure 1-2 shows an abridged version of this hierarchy.

Figure 1-2 Part of the Internet Domain Name System, augmented for Bonjour



At the top of the inverted tree is the root domain. Below it are some of the top-level domains: `com.`, `edu.`, and `org.`, and the local Bonjour "domain" `local.`, discussed further in "Bonjour and the Local Link" (page 17). Below the top level are a few second-level domains, `apple`, `darwin`, and `zeroconf`. The tree can extend infinitely downward with, for example, `www`, at the third level.

You may have noticed that the trailing dot is left off of most domain names. The trailing dot does, however, have meaning. A domain name ending in a trailing period, such as `www.apple.com.`, is known as a **fully qualified domain name**, much like an absolute path (such as `/usr/bin`) in a UNIX file system.

If you type `wibble.apple.com` into your Web browser (without the trailing dot), the system will treat it as an unqualified (partial) name and append the names from your list of search domains, such as `apple.com.`, `earthlink.net.`, `myschool.edu.`, etc. The system will first try to append `.` ("dot," the root domain), but if the name `wibble.apple.com.` doesn't exist, it will continue down the list and try `wibble.apple.com.apple.com.`, `wibble.apple.com.earthlink.net.`, `wibble.apple.com.myschool.edu.`, and so on, which is almost certainly not what you wanted.

Bonjour and the Local Link

Bonjour protocols deal in large part with the part of the network called the **local link**. A host's local link, or **link-local network**, includes itself and all other hosts that can exchange packets without IP header data being modified. In practice, this includes all hosts not separated by a router.

On Bonjour systems, `local.` is used to indicate a name that should be looked up using an IP multicast query on the local IP network.

Note that `local.` is not really a domain. You can think of `local.` as a pseudo-domain. It differs from conventional DNS domains in a fundamental way: names within other domains are globally unique; link-local domain names are not. There is only one logical DNS entry in the world named `www.apple.com.`, and because of the way DNS works, there can be only one. Host names ending in `local.`, on the other hand, are managed by a collection of multicast DNS responders on the local network, so the naming scope is just that: local. There can easily be two hosts named `meow.local.` in the world, or even the same building, just not on the same local network.

Globally unique names are important and useful—in fact, they are one of the significant achievements of the Internet—but they require a certain level of administrative effort to set up and maintain. Local names are useful only on the local network, but in cases where that is adequate, they provide a way to refer to network devices using names instead of IP numbers, and of course they require less effort and expense to coordinate than globally unique names.

Globally unique names are particularly useful on local networks that have no connection to the global Internet, either by design or because of interruption, and on small, temporary networks, such as a pair of computers linked by a crossover cable, or a few people playing network games using laptops on the wireless network of a home or cafe.

If a name collision on the local network occurs, a Bonjour host finds a new name automatically (in the case of a device without a screen) or by asking the user (in the case of a personal computer).

Bonjour Names for Existing Service Types

Bonjour service types are named according to the existing Internet standard for IP services. The Internet Assigned Numbers Authority (IANA) keeps a registry of TCP and UDP protocol names and ports assigned to each, and Bonjour services are named according to this list. The list used by Mac OS X can be found in `/etc/services`, and the most current version of the list is on the IANA website at http://www.iana.org/assignments/port-numbers.

In order to distinguish services from domain names in DNS resource records, components of service type names include underscore prefixes. The format for Bonjour service type names is thus

_ApplicationProtocolName . _TransportProtocolName .

The application protocol name is the official IANA-registered name for the protocol, for example, `ftp`, `http` or `printer`, as described above. The transport protocol name is either `tcp` or `udp`, depending on the transport protocol the service uses. An FTP service running over TCP would have a service type of `_ftp._tcp`. Services of this type would register DNS PTR records named `_ftp._tcp.local.` with their hosts' multicast DNS responders.

Bonjour Names for New Service Types

If you are designing a new protocol to advertise as a Bonjour network service, you should register it with IANA at http://www.iana.org or http://www.dns-sd.org/ServiceNames.html.

The IANA currently requires that every registered service be associated with a "well-known port" or range of well-known ports. For example, `http` is assigned port `80`, so that whenever you visit a website in your web browser, the application assumes that the HTTP service is running on port `80` unless you specify otherwise. This way, the port number for a website need only be memorized if the website is configured in a non-standard way.

With Bonjour, however, you don't have to know about port numbers. Because client applications can discover your service with a simple query for the service type, well-known ports are unnecessary. At some point in the future, IANA may begin registering protocol names without requiring an associated well-known port.

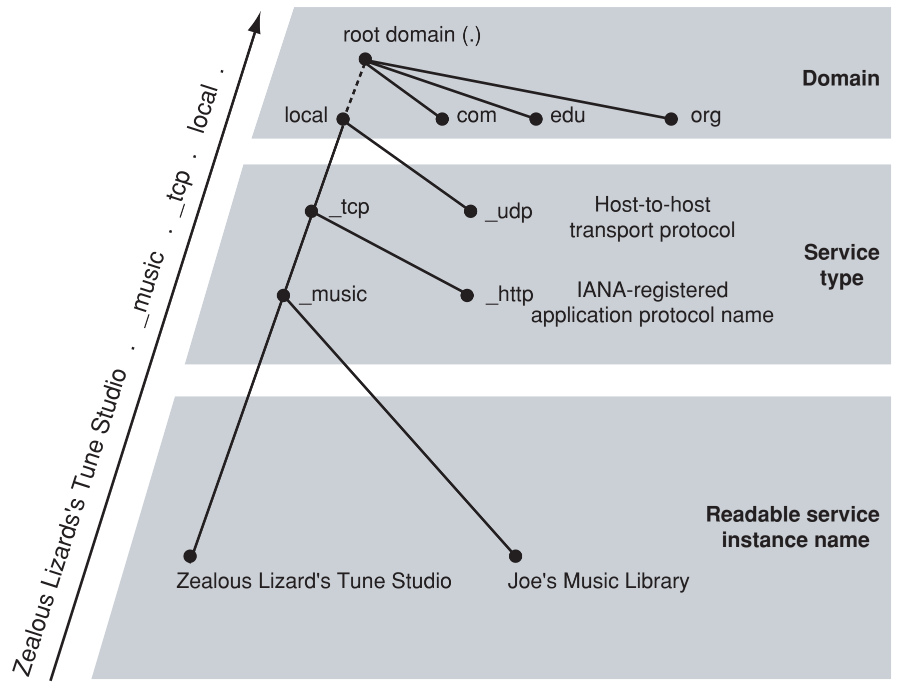
Bonjour Names for Service Instances

Service instance names are intended to be human-readable strings. As such, you should name them descriptively, and let the user override whatever default name you provide. Because they are intended to be browsed rather than typed, service instance names can be any Unicode string encoded with UTF-8, up to 63 octets (bytes) in length.

For example, an application for sharing music over the network might use the local user's name for a music sharing service, such as `Ed's Music Library`, by default. The user could override the default and name the service `Zealous Lizard's Tune Studio`, and the application would register a DNS SRV record named `Zealous Lizard's Tune Studio._music._tcp.local.`, assuming the application's music sharing protocol was associated with the name `music`.

Figure 1-3 (page 19) illustrates the organization of the name of a Bonjour service instance. At the top level of the tree is the domain, such as `local.` for the local network. Below the domain is the service type, which consists of the protocol name preceded by an underscore (_`music`) and the transport protocol, also preceded by an underscore (_`tcp`). At the bottom of the tree is the human-readable service instance name, such as `Zealous Lizard’s Tune Studio`. The complete name is a path along the tree from bottom to top, with each component separated by a dot.

Figure 1-3 Organization of a Bonjour service name


Zealous Lizards's Tune Studio . _music . _tcp . local .
root domain (.)
Domain
local
com
edu
org
_tcp
_udp
Host-to-host transport protocol
Service type
_music
_http
IANA-registered application protocol name
Readable service instance name
Zealous Lizard's Tune Studio
Joe's Music Library

DNSServiceDiscovery Tasks

The DNSServiceDiscovery API helps you to perform three main tasks:

- registering a service
- browsing for services
- resolving the current address of a service instance

In support of these main tasks, this API can directly assist you in performing two subsidiary tasks:

- enumerating domains (finding recommended service domains)
- updating registrations (changing your DNS registration data dynamically)

Before You Start

The next few paragraphs describe some things you should know about this API before attempting any of the tasks.

Most functions in this API do not return all of their data using their function return or parameter block. Instead, they require you to provide a callback function that can handle data sent asynchronously. This data is in the form of a reply type, such as a `DNSServiceRegistrationReply`. There are separate reply types for registration, enumeration, browsing, resolving addresses, and updating registration records. These reply types can be found in the section "Constants and Data Types" (page 43).

Your callback function may be called multiple times in response to a single function call on your part. For example, you might request a list of available services. Your callback would be called once for each available service that matches your request, then called again whenever a matching service is added or removed.

Some functions return error codes or status flags in the usual way, but many do not. In these cases, any error codes or status flags are sent to your callback function as part of the asynchronous reply, along with—or instead of—any returned data.

Most of the functions in this API use a common set of parameters to describe services. You will need to supply some or all of these parameters, depending on the purpose of your call. In many cases, you will provide some parameters, such as the domain and type of service, and your callback function will receive data corresponding to other parameters, such as the service name and IP address of a matching service.

Here is a list of the common parameters used to describe a service.

- Name—human readable name of the service, such as "`Sales Laser Printer`"
- Registration type—the type of service, such as "`_printer._tcp`"
- Domain—the domain for the service, typically "`local.`" but you can usually pass an empty string "" to specify the default domain

Note: The dot in "`local.`" is part of the domain name. It signifies that the domain is fully qualified, which prevents searching outside the local network or LAN segment.

- Port—the port number for the service
- Text record—an optional field containing any additional information that may be needed to use the service, such as a print queue name

Registering a Service

When your service starts up, you need to register with the mDNS responder daemon so that applications can discover your service. This section provides a general overview of the process, followed by a set of step-by-step instructions and some sample code.

Note: If you have created a new network service type, you need to contact the IANA to obtain a standard DNS resource for your new type.

Before registering your service, you need to create a network socket and obtain an IP address (this can be a link-local IP address or a universal IP address). Your service should be active and ready respond to service requests when you register.

The first step in registration is to allocate and initialize a `dns_service_discovery_ref` record for your service. This is an opaque type, so you call a create function to allocate it and initialize it with your chosen DNS name and service type. Your service type is passed as a standard DNS resource, as defined by the IANA. You can find the definition at http://www.iana.org/assignments/dns-parameters.

If the DNS name you've chosen is already in local use, you'll need to choose another name and try again, until you choose a locally-unique name.

When you register successfully, a Mach reply port is set up for your service.

The `dns_service_discovery_ref` record returned to you contains the Mach reply port address, which you extract using an accessor function.

You need to handle incoming Mach messages by passing them to a special message-handling function provided by this API. One way to do this is to generate a Mach reference that uses the message-handling function in a callback, create a runloop source from it, and add it to your `CFRunLoop`, (assuming you have one). In any event, you need to implement some method for passing incoming Mach messages to this message-handling function.

You have now registered your service; it is announced to the local network and its access information (IP address, port, and so on) can be found using multicast DNS, either by name or by browsing for services.

Here are the actual programming steps:

1. Allocate and initialize a `dns_service_discovery_ref` record by calling `DNSServiceRegistrationCreate`. You will be returned a `dns_service_discovery_ref` record and a Mach reply port will be set up.
2. Extract your Mach port address by calling `DNSServiceDiscoveryMachPort`.

3. Put a mechanism in place to receive Mach messages and route them to `DNSServiceDiscovery_handleReply`. If you are using a `CFRunLoop`, for example, you would do something like this:

Listing 2-1 Sample code for registration

```
// create a callback wrapper for the DNSServiceDiscovery_handleReply
// function
    static void MyHandleMachMessage(
                CFMachPortRef port, void *msg, CFIndex size, void *info)
    {
    DNSServiceDiscovery_handleReply(msg);
    }
// Create a port reference, use it to create a runloop source,
// and add it to the current runloop
static int AddDNSServiceClientToRunLoop(dns_service_discovery_ref client)
{
mach_port_t port = DNSServiceDiscoveryMachPort(client);
if (!port)
    return(-1);
else
    {
    CFMachPortContext  context    = { 0, 0, NULL, NULL, NULL };
    Boolean shouldFreeInfo;
    CFMachPortRef cfMachPort =
        CFMachPortCreateWithPort(kCFAllocatorDefault, port,
        MyHandleMachMessage, &context, &shouldFreeInfo);

    CFRunLoopSourceRef rls = CFMachPortCreateRunLoopSource(NULL,
        cfMachPort, 0);

    CFRunLoopAddSource(CFRunLoopGetCurrent(), rls,kCFRunLoopDefaultMode);
    CFRelease(rls);
    return(0);
    }
}
```

Note: If you are not using a `CFRunLoop`, you need to implement your own mechanism for receiving Mach messages and passing them to the `DNSServiceDiscovery_handleReply` routine.

Additional registration sample code can be found in the file `SamplemDNSClient.c`

Browsing for Network Services

Browsing for services using this API is fairly simple. You can find out what services of a given type are available in a given domain with a single function call.

To browse for available services, take the following step:

1. Call `DNSServiceBrowserCreate`, passing in the domain to search and the type of service you're interested in.

You can pass an empty string "" as the domain to browse—this automatically selects the default domain.

This function sends a separate reply to your callback function for every service instance that matches the specified type and domain, with additional calls when services are added or removed.

To create a list of available services, your application code must record each reply from `DNSServiceBrowserCreate`, concatenating them with sufficient logic to account for deletions.

To browse in multiple domains, or for multiple service types, make one call to `DNSServiceBrowserCreate` for each domain and service type of interest. Again, it is up to your application code to keep track of the replies.

Note: You can obtain a list of recommended domains to search by calling `DNSServiceDomainEnumerationCreate`. For details, see "Domain Enumeration" (page 25).

Don't disable the user interface or change the cursor while waiting for the replies from `DNSServiceBrowserCreate`. This task should be running in the background all the time. You normally call `DNSServiceBrowserCreate` only once per session. Whenever the list of services changes, data is sent to the callback function that you provide, so you can simply leave the callback active, and your list will always be up to date. This information typically changes infrequently, so the callback shouldn't use much CPU time.

Calling the browser-create function allocates a `dns_service_discovery_ref` record, so if you choose to deactivate your callback and repeat the search as needed, be sure to deallocate the record using `DNSServiceDiscoveryDeallocate` before calling `DNSServiceBrowserCreate` again. Otherwise, you will "leak" a record for every search.

The actual IP address and port of a given service instance are more ephemeral than the list of available services. You should resolve the current address of a service instance just prior to actually using the service, each time you use it. See the next section, "Resolving the Current Address of a Service Instance" (page 24).

Browsing sample code can be found in the file `SamplemDNSClient.c`

Resolving the Current Address of a Service Instance

Once you have the name, service type, and domain of a service, you can find the address (and any other access information you may need, such as a print queue name) by calling `DNSServiceResolverResolve`.

Important: Because service addresses can change dynamically, you should resolve the current address each time you use a service, just prior to actually using it.

To resolve the current address of a service instance, perform the following steps:

1. If you have not already done so, obtain a valid name, service type, and domain by calling `DNSServiceBrowserCreate`.

2. Call `DNSServiceResolverResolve`, passing in the service name, domain, and type. Your callback function will be called once, asynchronously, when the service address has been resolved.

Your callback will receive a `DNSServiceResolverReply` containing the current IP address, port, and other information you need to access the service.

One of the less obvious pieces of information you need is the IP address of the host computer's network interface. You need this because link-local addresses are not globally unique URLs—they are unique only on the local network. If the host computer has more than one network interface, such as an Ethernet card and an Airport card, it can be connected to more than one local network. If you know which interface is in use, you know which local network the service is on, and the link-local address is fully scoped.

Resolver sample code can be found in the file `SamplemDNSClient.c`

Domain Enumeration

While Bonjour is typically used to register and browse for services in the `local.` domain, it is also possible to register your service or browse for available services in other domains. You can obtain a list of recommended domains for registration or browsing by calling `DNSServiceDomainEnumerationCreate`.

To obtain a list of recommended domains, take the following step:

1. Call `DNSServiceDomainEnumerationCreate`, passing in a Boolean that tells the function whether you intend to register or browse. You will receive a list of recommended search domains, including the default domain (typically "`local.`"). The list is sent asynchronously to your callback function.

Your callback will be called for each recommended domain and whenever a domain is added or removed. Your application code is responsible for assembling these replies into a list.

Your callback will be passed a `DNSServiceDomainEnumerationReply` struct, which contains a domain name with flags indicating whether the domain should be added, removed, or made the default. Another flag indicates whether the list is now complete or more is coming.

Note: Even if the flag indicates that the list is complete, your callback will be called again if a domain is added or removed, or made the default.

The `DNSServiceDomainEnumerationCreate` function also allocates and returns a `dns_service_discovery_ref` record to serve as your client ID. You are responsible for deallocating it.

Note: You do not need to call the domain enumeration function to search or register in the default domain. Just pass an empty string "" as the domain parameter when registering or browsing.

Enumeration sample code can be found in the file `SamplemDNSClient.c`

Updating Your Registration Dynamically

You will probably never need to update your registration dynamically, as Bonjour automatically handles the common cases, such as waking, sleeping, shutting down, and changing IP addresses.

An exception would be the need to update the text record associated with a service. If a text field contains a queue name, for example, and the queue name changes, you would need to update the text record for the service.

To update your DNS information, call `DNSServiceRegistrationUpdateRecord`, passing in an updated DNS resource record. This function calls for a reference returned from `DNSServiceDiscoveryAddRecord`, but you can pass a zero in this field to update the record returned by `DNSServiceRegistrationCreate`, which is your primary record.

Registration update sample code can be found in the file `SamplemDNSClient.c`

Functions and Callbacks

This section describes the functions that make up the DNSServiceDiscovery API and provides prototypes of the callbacks you need to implement. The callbacks are required to handle the asynchronous replies that many of these functions generate. The functions and callbacks are organized into the following sections:

Functions by Task

Registration

These functions allow services to register with the mDNS responder. Once a service is registered, it can be found—either by name or by browsing for services of this type—using multicast DNS requests.

- `DNSServiceRegistrationCreate` (page 33)
- `MyDNSServiceRegistrationReply_handler` (page 39)

`DNSServiceRegistrationCreate` (page 33)
: Allocates and initializes a `dns_service_discovery_ref` record, sets up a Mach reply port for the service, and registers the service in the specified domain.

`MyDNSServiceRegistrationReply_handler` (page 39)
: Your application implements this callback function to handle replies from `DNSServiceRegistrationCreate`. The reply is a `DNSServiceRegistrationReply`.

Mach Port Accessor

These functions help the applications programmer establish a connection with a Mach reply port.

- `DNSServiceDiscoveryMachPort` (page 31)
- `DNSServiceDiscovery_handleReply` (page 31)

DNSServiceDiscoveryMachPort (page 31)
  Returns a Mach reply port.

DNSServiceDiscovery_handleReply (page 31)
  Handles messages from your Mach port.

Service Browser

These functions allow applications to discover network services by browsing:

- DNSServiceBrowserCreate (page 29)
- MyDNSServiceBrowserReply_handler (page 36)

DNSServiceBrowserCreate (page 29)
  Returns a list of all network services of a specified type in a specified domain.

MyDNSServiceBrowserReply_handler (page 36)
  Your application implements this callback function to handle replies from `DNSServiceBrowserCreate`. The reply is a `DNSServiceBrowserReply`.

Service Address Resolver

These functions resolve the current IP address and port of a given service instance.

- DNSServiceResolverResolve (page 35)
- MyDNSServiceResolverReply_handler (page 40)

DNSServiceResolverResolve (page 35)
  Finds the current IP address and port for a service instance, as returned to your `DNSServiceBrowserReply_handler` callback function after a call to `DNSServiceBrowserCreate`.

MyDNSServiceResolverReply_handler (page 40)
  Your application implements this callback function to handle replies from `DNSServiceResolverResolve`.

Domain Enumeration

These functions list the recommended and default domains for registration or browsing:

- DNSServiceDomainEnumerationCreate (page 32)
- MyDNSServiceDomainEnumerationReply_handler (page 38)

DNSServiceDomainEnumerationCreate (page 32)
  Lists the recommended domains in which to register a service or browse for services.

MyDNSServiceDomainEnumerationReply_handler (page 38)
  Your application implements this callback function to handle replies from `DNSServiceDomainEnumerationCreate`. The reply is a `DNSServiceDomainEnumerationReply`.

Reference Record Deallocation

This function described in this section deallocates a `dns_service_discovery_ref` record and disconnects any associated Mach port.

- `DNSServiceDiscoveryDeallocate` (page 30)

`DNSServiceDiscoveryDeallocate` (page 30)
: Deallocates a `dns_service_discovery_ref` record and closes the connection to the server.

Dynamic Update

These functions provide the ability to add to, update, or delete registration information stored in the mDNS responder dynamically, after you have registered and while your service is running. An example would be updating the text record associated with a printer to indicate a change in printing capabilities. Bonjour automatically handles most update tasks, such as sleep/wake and location changes, so these functions are rarely needed.

- `DNSServiceRegistrationAddRecord` (page 32)
- `DNSServiceRegistrationRemoveRecord` (page 34)
- `DNSServiceRegistrationUpdateRecord` (page 35)

`DNSServiceRegistrationAddRecord` (page 32)
: Request that the mDNS Responder append an additional record to the DNS resource information associated with your service.

`DNSServiceRegistrationRemoveRecord` (page 34)
: Request that the mDNS responder remove a record from your service's registration information.

`DNSServiceRegistrationUpdateRecord` (page 35)
: Request the mDNS responder to update a DNS record for your service. Most services will never need to do this.

Functions

DNSServiceBrowserCreate

Returns a list of all network services of a specified type in a specified domain.

```
dns_service_discovery_ref DNSServiceBrowserCreate(
const char *regtype,
const char *domain,
DNSServiceBrowserReply callBack,
void *context);
```

Parameters

`regtype`
: The type of service, as specified by the IANA; `_printer._tcp` is an example.

domain
: The domain to search for the specified type of service; `local.` is an example. Pass an empty string "" to search the default domain.

callBack
: The `DNSServiceBrowserReply_handler` function to be called when a service has been found or removed. See "MyDNSServiceBrowserReply_handler" (page 36).

context
: A user-specified context that will be passed to the callback function.

Return Value
This function allocates and returns a `dns_service_discovery_ref` record. The caller is responsible for deallocating the record.

Discussion
Use this function to browse for a list of available services of a given type in a given domain. The requested service data is sent asynchronously to your callback function as a `DNSServiceBrowserReply`. Your callback may receive a number of calls asynchronously. Do not update your user interface (list of available services) while the DNSServiceBrowserReply has the "More" flag set, but do not disable the user interface or change the cursor while waiting for data. Your callback should remain active as long as your service browser is running. To use a service found by this function, obtain the actual address information for that service by calling `DNSServiceResolverResolve`. To search for multiple service types, or to search multiple domains, make multiple calls to `DNSServiceBrowserCreate`.

Sample code: `SamplemDNSClient.c`

Version Notes
Introduced in OS X version 10.2.

DNSServiceDiscoveryDeallocate

Deallocates a `dns_service_discovery_ref` record and closes the connection to the server.

```
void DNSServiceDiscoveryDeallocate(
dns_service_discovery_ref dnsServiceDiscovery);
```

Parameters

dnsServiceDiscovery_ref
: A `dns_service_discovery_ref` record as returned from calling `DNSServiceRegistrationCreate`, `DNSServiceBrowserCreate`, or `DNSServiceDomainEnumerationCreate`.

Return Value
None.

Discussion
You are responsible for deallocating `dns_service_discovery_ref` records returned by creation or enumeration calls. For registration or browsing, you typically make a single creation call, and the returned record remains active until your application terminates. In these cases, it is not necessary to deallocate the record. The enumeration function can also be called once, and the callback left active, or it can be called as needed. In the latter case, you should deallocate the record before calling the enumeration function again to prevent a memory leak.

A Mach reply port is set up for your application when you call `DNSServiceRegistrationCreate`. Deallocating the returned `dns_service_discovery_ref` disconnects your application from the Mach port and unregisters your service. You can use this technique if you need to unregister and re-register your service without shutting down your application.

Sample code: `SamplemDNSClient.c`

Version Notes
Introduced in OS X version 10.2.

DNSServiceDiscoveryMachPort

Returns a Mach reply port.

```
mach_port_t DNSServiceDiscoveryMachPort(
dns_service_discovery_ref dnsServiceDiscovery);
```

Parameters

`registration`

A `dns_service_discovery_ref` a s returned from `DNSServiceRegistrationCreate`.

Return Value
This function returns a Mach reply port. A `NULL` value indicates that no address was specified or some other error occurred which prevented the resolution from being started.

Discussion
This function returns a Mach reply port which will be sent messages as appropriate. These messages should be handled by the `DNSServiceDiscovery_handleReply` function, which needs to be integrated into your run loop. See `DNSServiceDiscovery_handleReply` (page 31).

Sample code: `SamplemDNSClient.c`

Version Notes
Introduced in OS X version 10.2.

DNSServiceDiscovery_handleReply

Handles messages from your Mach port.

```
void DNSServiceDiscovery_handleReply(void *replyMsg);
```

Parameters

`*replyMsg`

The Mach message.

Discussion
Install this function as a callback to handle messages from your Mach port if you are providing a network service and using the mDNS responder.

Sample code: `SamplemDNSClient.c`

Version Notes
Introduced in OS X version 10.2.

DNSServiceDomainEnumerationCreate

Lists the recommended domains in which to register a service or browse for services.

```
dns_service_discovery_ref DNSServiceDomainEnumerationCreate(
int registrationDomains,
DNSServiceDomainEnumerationReply callBack,
void *context);
```

Parameters

registrationDomains

A Boolean indicating whether you are looking for recommended registration domains (equivalent to the AppleTalk zone list in the AppleTalk Control Panel) or recommended browsing domains (equivalent to the AppleTalk zone list in the Chooser).

callBack

The `DNSServiceDomainEnumerationReply_handler` function to be called when domains are found or removed See "`MyDNSServiceDomainEnumerationReply_handler`" (page 38).

context

A user-specified context that will be passed to the callback function.

Return Value

This function allocates and returns a `dns_service_discovery_ref` record. The caller is responsible for deallocating the record. This record does not contain the requested domain data—that information is sent to your callback function asynchronously.

Discussion

You can use this function obtain a list of recommended domains in which to register your service or a list of recommended domains in which to browse for services. This is not necessary if you want to simply register in or search the default domain. You can pass an empty string "" as the domain parameter for the registration and browser functions, and they will use the default domain without requiring you to look it up. Use this function if you need to do something more complex.

Sample code: `SamplemDNSClient.c`

Version Notes

Introduced in OS X version 10.2.

DNSServiceRegistrationAddRecord

Request that the mDNS Responder append an additional record to the DNS resource information associated with your service.

```
DNSRecordReference DNSServiceRegistrationAddRecord(
dns_service_discovery_ref dnsServiceDiscovery,
uint16_t rrtype,
uint16_t rdlen,
const char *rdata,
uint32_t ttl);
```

Parameters

dnsServiceDiscovery

The `dns_service_discovery_ref` for your service, as returned from a `DNSServiceRegistrationCreate` call.

rrtype

A standard DNS Resource Record Type, as defined at `http://www.iana.org/assignments/dns-parameters`.

rdlen

Length of the data block to follow (*rdata*).

rdata

Opaque binary Resource Record data—up to 64 kB of whatever you need to store.

ttl

Time-to-live for the added record. The TTL can be set to any signed 32-bit value.

Return Value

A `DNSRecordReference`, an opaque reference that can be passed to `DNSServiceRegistrationUpdateRecord` or `DNSServiceRegistrationRemoveRecord` to update or remove this record. If an error occurs, this value will be zero or negative.

Discussion

This function appends a resource record to your existing DNS service registry. This should not be necessary for existing services.

Sample code: `SamplemDNSClient.c`

Version Notes

Introduced in OS X version 10.2.

DNSServiceRegistrationCreate

Allocates and initializes a `dns_service_discovery_ref` record, sets up a Mach reply port for the service, and registers the service in the specified domain.

```
dns_service_discovery_ref DNSServiceRegistrationCreate(
const char *name,
const char *regtype,
const char *domain,
uint16_t port,
const char *txtRecord,
DNSServiceRegistrationReply callBack,
void *context);
```

Parameters

name

The name of this service instance (e.g. "Steve's Printer").

regtype

The service type, such as "`_printer._tcp.`" For the service type specifications, see RFC 2782 (DNS SRV).

domain

The domain in which to register the service, such as "`local.`" Set this parameter to "" (empty string) to use the default domain(s).

port

The local IP port on which this service is being offered (in network byte order).

txtRecord
: Optional protocol-specific additional information. This parameter is provided to support legacy protocols that have no capability negotiation and require text configuration, such as LPR printing. If you are creating a new protocol, please not use text records.

callBack
: The `DNSServiceRegistrationReply_handler` callback function to be called with any flags or errors. See "`MyDNSServiceRegistrationReply_handler`" (page 39).

context
: A user specified context which will be passed to the callback function.

Return Value
Allocates, initializes, and returns a `dns_service_discovery_ref` record, an opaque data structure. Use this record to obtain your Mach port address from `DNSServiceDiscoveryManPort`. This record is also used to identify your service to other functions in this API. You are responsible for deallocating the returned record when you are finished with it.

Discussion
Use this function to create register a service. Calling this function also sets up a Mach reply port for your service. To obtain the Mach port address, pass the returned `dns_service_discovery_ref` record to `DNSServiceDiscoveryMachPort`. Deallocating the returned record also closes your connection to the Mach reply port. Your service needs a network socket and an IP address, and should be ready to respond to service requests, when you call this function.

Sample code: `SamplemDNSClient.c`

Version Notes
Introduced in OS X version 10.2.

DNSServiceRegistrationRemoveRecord

Request that the mDNS responder remove a record from your service's registration information.

```
DNSServiceRegistrationReplyErrorType DNSServiceRegistrationRemoveRecord(
dns_service_discovery_ref ref,
DNSRecordReference reference);
```

Parameters

ref
: A `dns_service_discovery_ref` as returned from a `DNSServiceRegistrationCreate` call

reference
: A `dnsRecordReference` as returned from calling `DNSServiceRegistrationAddRecord`. To remove your primary record, as returned from DNSServiceRegistrationCreate, pass a zero in this parameter.

Return Value
A `DNSServiceRegistrationReplyErrorType.` If an error occurs, this value will be nonzero.

Discussion
Call this function to remove a record from your service's DNS registration information. You do not need to do this in the ordinary course of events. If you remove your primary record, you effectively unregister your service, but this is a side effect, not a design feature—it does not deallocate your registration record or disconnect your Mach port, for example.

Sample code: `SamplemDNSClient.c`

Version Notes
Introduced in OS X version 10.2.

DNSServiceRegistrationUpdateRecord

Request the mDNS responder to update a DNS record for your service. Most services will never need to do this.

```
DNSServiceRegistrationReplyErrorType DNSServiceRegistrationUpdateRecord(
dns_service_discovery_ref ref,
DNSRecordReference reference,
uint16_t rdlen,
const char *rdata,
uint32_t ttl);
```

Parameters

ref
A `dns_service_discovery_ref` as returned by `DNSServiceRegistrationCreate`.

reference
A `dnsRecordReference` as returned by `DNSServiceRegistrationAddRecord`. To update information in your primary record, as returned from, set this parameter to zero. You might do this, for example, if you need to update the text field for a legacy protocol while a service is running.

rdlen
Length of the data block to follow (*rdata*).

rdata
Opaque binary resource record data, containing up to 64 kB of whatever data you choose.

ttl
Time to live for the updated record. The TTL can be set to any signed 32-bit value.

Return Value
A value of type `DNSServiceRegistrationReplyErrorType`. If an error occurs, this value will be non-zero.

Discussion
You might use this function to update a text record associated with your service, if you are supporting a legacy protocol and the information stored in the text record changes. Otherwise, you are unlikely to have use for this function.

Sample code: `SamplemDNSClient.c`

Version Notes
Introduced in OS X version 10.2.

DNSServiceResolverResolve

Finds the current IP address and port for a service instance, as returned to your `DNSServiceBrowserReply_handler` callback function after a call to `DNSServiceBrowserCreate`.

```
dns_service_discovery_ref DNSServiceResolverResolve(
const char *name,
const char *regtype,
const char *domain,
DNSServiceResolverReply callBack,
void *context);
```

Parameters

name
: The name of the service instance, as returned from `DNSServiceBrowserCreate`.

regtype
: The type of service, as returned from `DNSServiceBrowserCreate`.

domain
: The domain in which to find the service, as returned from `DNSServiceBrowserCreate`.

callBack
: The `DNSServiceResolverReply_handler` function to be called when the specified address has been resolved. See "MyDNSServiceResolverReply_handler" (page 40).

context
: A user specified context which will be passed to the callback function.

Return Value
This function returns a `dns_service_discovery_ref` record, an opaque data type. The requested service address data is sent to your callback function asynchronously as a `DNSServiceResolverReply`.

Discussion
Use this function to resolve the current IP address of a service after you have obtained its name, service type, and domain from `DNSServiceBrowserCreate`. The IP address can change dynamically, and services can be removed or unplugged at any time, so call this function immediately before using any network service, and call it each time you use the service.

Sample code: `SamplemDNSClient.c`

Version Notes
Introduced in OS X version 10.2.

MyDNSServiceBrowserReply_handler

Your application implements this callback function to handle replies from `DNSServiceBrowserCreate`. The reply is a `DNSServiceBrowserReply`.

```
static void MyDNSServiceBrowserReply_handler(
DNSServiceBrowserReplyResultType resultType,
const char *replyName,
const char *replyType,
const char *replyDomain
DNSServiceDiscoveryReplyFlags flags,
void *context)
```

Parameters

resultType
: The `DNSServiceBrowserReplyResultType`. Possible values are `DNSServiceBrowserReplyAddInstance` and `DNSServiceBrowserReplyRemoveInstance`.

replyName
: The name of the service instance; `Sales Printer` is an example.

replyType
: The type of service; `_printer._tcp` is an example.

replyDomain
: The domain of the service; `local.` is an example.

flags
: Any error codes or flags. See `DNSServiceDiscoveryFlags` in "Constants and Data Types" (page 43). If there are multiple known services, the `kDNSServiceDiscoveryMoreRepliesImmediately` flag will be set until you have received replies for all of them. Do not update the user interface (list of services) while this flag is set, but do not disable the user interface or change the cursor while waiting for this flag to clear.

context
: A user-specified context that will be passed to the callback function.

Return Value

Not applicable.

Discussion

This is a prototype for your callback function, to be passed in to `DNSServiceBrowserCreate`. Your callback function will be called once for every service instance of the specified type found in the domain. It will be called again if new instances are added to the domain or known instances are removed. The sum of these replies is the current list of service instances. Note that you receive individual records; your application is responsible for building and maintaining a list.

This callback should remain active as long as your service browser is running. That way the list of available services will always be up to date.

Your callback is sent a `DNSServiceBrowserReply` record, a structure containing the reply type (add or delete a service instance from your list), instance name, service type, and domain, any flags or errors, and a context that will be returned to the callback function. A simple code sample follows.

```
static void browse_reply(DNSServiceBrowserReplyResultType resultType,
    const char *replyName,
    const char *replyType,
    const char *replyDomain
    DNSServiceDiscoveryReplyFlags flags,
    void *context)
{
char *op =
(resultType == DNSServiceBrowserReplyAddInstance) ? Found" : "Removed";
printf("Service \"%s\", type \"%s\", domain \"%s\" %s", replyName, replyType,
 replyDomain, op);
if (flags) printf(" Flags: %X", flags);
printf("\n");
}
```

Sample code: `SamplemDNSClient.c`

Version Notes

Introduced in OS X version 10.2.

MyDNSServiceDomainEnumerationReply_handler

Your application implements this callback function to handle replies from `DNSServiceDomainEnumerationCreate`. The reply is a `DNSServiceDomainEnumerationReply`.

```
static void regdom_reply(
DNSServiceDomainEnumerationReplyResultType resultType,
const char *replyDomain,
DNSServiceDiscoveryReplyFlags flags,
void *context)
```

Parameters

resultType
The `DNSServiceDomainEnumerationReplyResultType`. Possible values are `DNSServiceDomainEnumerationReplyAddDomain`, `DNSServiceDomainEnumerationReplyAddDomainDefault`, or `DNSServiceDomainEnumerationReplyRemoveDomain`.

replyDomain
A domain in which to register or browse; `local.` is an example.

flags
Any error codes or flags. See `DNSServiceDiscoveryFlags` in "Constants and Data Types" (page 43). If there are multiple recommended domains, the `kDNSServiceDiscoveryMoreRepliesImmediately` flag will be set until you have received replies for all of them.

context
A user-specified context that will be passed to the callback function.

Return Value
Not applicable.

Discussion
This is a prototype for your callback function, to be passed in to `DNSServiceDomainEnumerationCreate`. Your callback function will be called once for every recommended domain. It will be called again if domains are added or removed, or if the default changes. The sum of these replies is the current list of recommended domains.

Your callback is sent a `DNSServiceDomainEnumerationReply` record, a struct containing the reply type (add or delete a domain from your list), a domain name, any flags or errors, and a context that will be returned to the callback function. A code sample follows.

```
#define DomainMsg(X) (
(X) == DNSServiceDomainEnumerationReplyAddDomain
? "Added"                :        \
(X) == DNSServiceDomainEnumerationReplyAddDomainDefault
? "(Default)"                :        \
(X) == DNSServiceDomainEnumerationReplyRemoveDomain
? "Removed"              : "Unknown")
static void regdom_reply(
    DNSServiceDomainEnumerationReplyResultType resultType,
    const char *replyDomain,
    DNSServiceDiscoveryReplyFlags flags,
    void *context)
    {
    printf("Recommended Registration or Browse Domain %s %s", replyDomain,
DomainMsg(resultType));
```

```
    if (flags) printf(" Flags: %X", flags);
    printf("\n");
    }
```

Sample code: `SamplemDNSClient.c`

Version Notes
Introduced in OS X version 10.2.

MyDNSServiceRegistrationReply_handler

Your application implements this callback function to handle replies from `DNSServiceRegistrationCreate`. The reply is a `DNSServiceRegistrationReply`.

```
static void reg_reply(
DNSServiceRegistrationReplyErrorType errorCode,
void *context)
```

Parameters

`errorCode`

Any error codes or flags. See `DNSServiceDiscoveryFlags` in "Constants and Data Types" (page 43).

`context`

A user-specified context that will be passed to the callback function.

Return Value
Not applicable.

Discussion
This is a prototype for your callback function, to be passed in to `DNSServiceRegistrationCreate`.

Your callback is sent a `DNSServiceRegistrationReply`, consisting of either no flags, a message conflict flag, or an error code. If your requested DNS name is already in use, choose another name and re-register. For example, your default service name may be in use by another copy of your service on the same network, such as an identical printer. If your service has no user interface for naming, it is good practice to append a number to your default name and retry, incrementing the number as necessary to obtain a unique name. A code sample for a service with a user interface follows.

```
static void reg_reply(
    DNSServiceRegistrationReplyErrorType errorCode,
    void *context)
    {
    printf("Got a reply from the server: ");
    switch (errorCode)
        {
        case kDNSServiceDiscoveryNoError:
            printf("Name now registered and active\n"); break;
        case kDNSServiceDiscoveryNameConflict:
            printf("Name in use, please choose another\n"); exit(-1);
        default:
        }
            printf("Error %d\n", errorCode); return;
```

Sample code: `SamplemDNSClient.c`

Version Notes
Introduced in OS X version 10.2.

MyDNSServiceResolverReply_handler

Your application implements this callback function to handle replies from `DNSServiceResolverResolve`.

```
static void MyDNSServiceResolverReply_handler(struct
sockaddr *interface,
struct sockaddr *address,
const char *txtRecord,
DNSServiceDiscoveryReplyFlags flags,
void *context)
```

Parameters

`interface`
A `sockaddr` containing the IP address of the host's network interface. If the host is connected to more than one local network, such as an ethernet LAN and a wireless LAN, this identifies which network contains the service.

`address`
A *`sockaddr`* containing the link-local IP address and port number of the service.

`txtrecord`
A character string containing any additional information, such as a print queue name. If there is no additional information, this is an empty string.

`flags`
Any error codes or flags. See `DNSServiceDiscoveryFlags` in "Constants and Data Types" (page 43).

`context`
A user-specified context that will be passed to the callback function.

Return Value
Not applicable.

Discussion
This is a prototype for your callback function, to be passed in to `DNSServiceResolverResolve`. Your function will be called when the service address has been resolved. Your callback will receive a `DNSServiceResolverReply` record. This struct contains the host interface IP address, the service IP address and port, an optional text record with any additional demultiplexing information, any flags or errors, and a context that will be returned to the callback function.

A code sample follows.

```
static void resolve_reply(struct sockaddr *interface,
    struct sockaddr *address,
    const char *txtRecord,
    DNSServiceDiscoveryReplyFlags flags,
    void *context)
{
if (address->sa_family (!= AF_INET)
    printf("Unknown address family %d\n", address->sa_family);
else
    {
    struct sockaddr_in *ip = (struct sockaddr_in *)address;
    union { uint32_t l; u_char b[4]; } addr = { ip->sin_addr.s_addr };
    union { uint16_t s; u_char b[2]; } port = { ip->sin_port };
    uint16_t PortAsNumber = ((uint16_t)port.b[0]) << 8 | port.b[1];
    const char *src = txtRecord;
    printf("Service can be reached at %d.%d.%d.%d:%u", addr.b[0],
                     addr.b[1], addr.b[2], addr.b[3], PortAsNumber);
    while (*src)
        {
        char txtInfo[256];
        char *dst = txtInfo;
        const char *const lim = &txtInfo[sizeof(txtInfo)];
        while (*src && *src != 1 && dst < lim-1) *dst++ = *src++;
        *dst++ = 0;
        printf(" TXT \"%s\"", txtInfo);
        if (*src == 1) src++;
        }
    if (flags) printf(" Flags: %X", flags);
    printf("\n");
    }
}
```

Sample code: `SamplemDNSClient.c`

Version Notes
Introduced in OS X version 10.2.

Legacy Document | 2005-04-29 | © 2001, 2005 Apple Computer, Inc. All Rights Reserved.

Constants and Data Types

This section describes the constants and data types used by the DNSServiceDiscovery API:

DNSServiceDiscovery Flags and Errors

These flags and data types are used by most DNSServiceDiscovery functions and callbacks.

```
/* Opaque internal data type */
typedef struct _dns_service_discovery_t * dns_service_discovery_ref;

/* possible reply flags values */
enum {
    kDNSServiceDiscoveryNoFlags= 0,
    kDNSServiceDiscoveryMoreRepliesImmediately= 1 << 0,
};
// If the kDNSServiceDiscoveryMoreRepliesImmediately flag is set,
// do not update your UI

/* possible error code values */
typedef enum
{
kDNSServiceDiscoveryWaiting     = 1,
kDNSServiceDiscoveryNoError     = 0,
// mDNS Error codes are in the range
// FFFE FF00 (-65792) to FFFE FFFF (-65537)
kDNSServiceDiscoveryUnknownErr        = -65537,       // 0xFFFE FFFF
kDNSServiceDiscoveryNoSuchNameErr     = -65538,
kDNSServiceDiscoveryNoMemoryErr       = -65539,
kDNSServiceDiscoveryBadParamErr       = -65540,
kDNSServiceDiscoveryBadReferenceErr   = -65541,
kDNSServiceDiscoveryBadStateErr       = -65542,
kDNSServiceDiscoveryBadFlagsErr       = -65543,
kDNSServiceDiscoveryUnsupportedErr    = -65544,
kDNSServiceDiscoveryNotInitializedErr = -65545,
kDNSServiceDiscoveryNoCache           = -65546,
kDNSServiceDiscoveryAlreadyRegistered = -65547,
kDNSServiceDiscoveryNameConflict      = -65548,
kDNSServiceDiscoveryInvalid           = -65549,
kDNSServiceDiscoveryMemFree           = -65792        // 0xFFFE FF00
} DNSServiceRegistrationReplyErrorType;
```

Registration and Record Update

These types are used for DNS registration and update.

```
typedef void (*DNSServiceRegistrationReply) (
DNSServiceRegistrationReplyErrorType errorCode,
void *context
);

typedef uint32_t DNSRecordReference;
```

Resolver

These types and flags are used when resolving the IP address for a service.

```
typedef void (*DNSServiceResolverReply) (
    struct sockaddr *interface,
        // Host interface IP addr--needed if multiple LAN connections
    struct sockaddr *address,
        // Service link-local IP address, including port number
    const char *txtRecord,
    DNSServiceDiscoveryReplyFlags flags,
    void *context
);
```

Domain Enumeration

These constants are used when determining the domain(s) in which you should search for or register a service

```
typedef enum
{
    DNSServiceDomainEnumerationReplyAddDomain, // Domain found
    DNSServiceDomainEnumerationReplyAddDomainDefault,
        // Domain found (and should be selected by default)
    DNSServiceDomainEnumerationReplyRemoveDomain,
        // Domain has been removed from network
} DNSServiceDomainEnumerationReplyResultType;
typedef enum
{
    DNSServiceDiscoverReplyFlagsFinished,
    DNSServiceDiscoverReplyFlagsMoreComing,
} DNSServiceDiscoveryReplyFlags;
typedef void (*DNSServiceDomainEnumerationReply) (
    DNSServiceDomainEnumerationReplyResultType resultType,
    const char *replyDomain,
    DNSServiceDiscoveryReplyFlags flags,
    void *context
);
```

Service Browser

These types and constants are used when browsing for services

```
typedef enum
{
    DNSServiceBrowserReplyAddInstance,
        // Instance of service found
    DNSServiceBrowserReplyRemoveInstance
        // Instance has been removed from network
} DNSServiceBrowserReplyResultType;
typedef void (*DNSServiceBrowserReply) (
    DNSServiceBrowserReplyResultType resultType, // Add or remove
    const char *replyName,
    const char *replyType,
    const char *replyDomain,
    DNSServiceDiscoveryReplyFlags flags,
    void *context
);
```

Legacy Document | 2005-04-29 | © 2001, 2005 Apple Computer, Inc. All Rights Reserved.

Document Revision History

This table describes the changes to *DNSServiceDiscovery Mach-Based API*.

Date	Notes
2005-04-29	Changed "Rendezvous" to "Bonjour."
2004-02-01	Corrected information about the TTL value for `DNSServiceRegistrationAddRecord` and `DNSServiceRegistrationUpdateRecord`. Synchronized information in Introduction and Bonjour Overview chapters with information in the socket-based DNSServiceDiscovery API document.

Legacy Document | 2005-04-29 | © 2001, 2005 Apple Computer, Inc. All Rights Reserved.