

Things that just work

*Covers Apple's
Bonjour APIs*



Zero Configuration Networking

The Definitive Guide

O'REILLY®

Daniel H. Steinberg & Stuart Cheshire

Using the C APIs

In Chapter 6, you experimented with advertising, browsing for, and looking up Zeroconf services from the command line. In this chapter, you will learn how to perform those same operations programmatically, using the C APIs. The remaining chapters in the book address other APIs and languages. In each of the APIs, the general concepts are the same—you perform one of the basic DNS-SD operations and then receive results asynchronously. In C, this means you initiate a DNS-SD action such as browsing and provide the address of a callback function. When there is a response, the callback function is called and the appropriate information is passed to it. By the end of this chapter, you could write your own version of the *dns-sd* command-line tool.

If you skipped the preceding chapter, you may want to go back and read it, since the *dns-sd* command-line tool is a good learning tool for exploring the concepts, and it's a good debugging aid if your code is not working as expected. If you are creating a program that registers a service, then you can browse with the command-line tool to verify that the service is being advertised. As you run and quit your growing application during development, you can leave the command-line browser running in a terminal window to confirm when you have successfully registered your service. Similarly, if you are building an application that needs to browse for services of a particular type, and you don't have any real instances of that service handy, then you can use the command-line tool to register pretend instances for your application to discover.

Asynchronous Programming Model

Before diving into the details of Zeroconf programming, there's one important high-level issue to understand—DNS-SD operations are asynchronous. That means that when you initiate some DNS-SD operation, the results of that operation may happen immediately, half a second later, a minute later, or even days later. Consider one real-world example, iTunes. When you run iTunes, it browses for other music sources on the local network. If you launch iTunes at home, there may be no other

music sources, so nothing appears. A few days later, a friend may come to visit with a laptop offering shared iTunes music. Your friend connects his laptop to your network and, instantly, he appears in your iTunes list. That browse operation initiated by iTunes several days ago has just yielded its first result!

To create a good user experience, it's essential to understand and embrace this asynchronous programming philosophy. We've all seen, at some time or another, bad user interfaces where you get to watch some cute animation for 15 seconds while the computer "searches" the network, and then it stops "searching" and shows you what it's found. If the thing you're looking for isn't there, you have to try to fix what's wrong, then click some refresh button and watch the cute animation for another 15 seconds before you find out whether it worked this time.

For the programmer, there's no right answer for a fixed search time. If you set the search time too short, you risk giving up before all devices have responded, particularly on slow or busy networks. If you set the search time too long, you frustrate the user. A 15-second wait may be tolerable just once, but when the user is trying to troubleshoot a problem and she has to suffer that 15-second wait *over and over and over*, she gets very frustrated. DNS-SD solves this dilemma by simply not having a fixed search time. DNS-SD operations run for an indeterminate length of time, until you stop them. Results are delivered as they come, not all at the end after some fixed timeout. You can expect the first browsing results to show up in as little as a few milliseconds. Slower devices may take a second or two to respond. Devices not yet connected to the network will, of course, only show up once they are connected, which could be hours, days, or weeks later.

Don't be tempted to make a Zeroconf application that starts a browse operation, lets it run for some fixed time, and then stops it and presents the results to the user. Zeroconf applications don't work this way, and yours will look crude and amateurish if you make this mistake. In particular, you should never have a Refresh button in Zeroconf UI—as long as a DNS-SD browsing list is on the screen, it should be fresh all the time, so the user never needs to refresh it. DNS-SD uses a range of techniques to make long-lived operations efficient on the network. Writing code that once a minute does a DNS-SD browse for five seconds to see what's there and then stops it would actually be a lot less efficient on the network than just leaving a browse operation running continuously.

To support this asynchronous programming philosophy, DNS-SD needs a way to deliver these asynchronous events to your application as they happen. There is no one universal mechanism for delivering asynchronous events that's common among all programming languages, programming models, and operating systems. Some programming models are built around a single main event loop that receives an event notification, handles it, receives another event notification, handles it, and so on. Other programmers prefer to write multithreaded code, where each thread blocks and waits for a particular event of interest. Still other models use concepts like signal

handlers or interrupt routines, where, when an event happens, the main thread is interrupted and suspended wherever it happens to be, the event handler routine is run, and then, afterward, control returns to the main thread.

Rather than try to dictate a particular programming model to use, the DNS-SD C API instead provides the necessary primitives so you can integrate its event delivery into whatever event-handling model you've chosen for your program. If you're using a main event loop, there are two things you need to do: you need to tell your main event loop to pay attention to DNS-SD event sources and, when one of those events happens, you need to ask DNS-SD what to do about it.

In this chapter, you'll see how to use DNS-SD C API with the various commonly used main event loop models: the Unix `select()` call, the Cocoa `RunLoop`, the Core Foundation `CFRunLoop`, and the Microsoft Windows `GetMessage()` Message Loop. You'll also see how to use DNS-SD C API by creating a separate independent thread for each active DNS-SD operation, if that's your preferred way of working.

We'll start by showing how to do each of the common DNS-SD operations using a Unix `select()` loop and then, later in the chapter, cover just the differences when using other event-handling models.

Event Handling with a `select()` Loop

It is important to understand the structure for working with the socket-based DNS-ServiceDiscovery APIs. First, you call a function that might, for example, initiate browsing for a service or register a service. Along with the other parameters, you pass in the address of an uninitialized `DNSServiceRef` (which the call initializes) and also pass in the address of the function that should be called back when interesting events happen.

You then call `DNSServiceRefSockFD()`, passing in the newly initialized `DNSServiceRef`, to extract the file descriptor for the Unix Domain Socket connection to the `mdnsd` daemon running in the background, and add this file descriptor to your `select()` loop. When the `mdnsd` daemon sends a message over the Unix Domain Socket connection to your process, your `select()` call will wake up, indicating that there is data waiting to be read on the file descriptor. You then call `DNSServiceProcessResult()`; the DNS-SD code decodes the message and calls the appropriate callback function you previously specified when starting the operation.

This section covers the three functions used to access sockets and to perform the callback: `DNSServiceRefSockFD()`, `DNSServiceProcessResult()`, and `DNSServiceRefDeallocate()`. A listing of the `DNSServiceDiscovery` Error Codes is also provided.

Event Callbacks

The use of asynchronous callbacks is essential for DNS-SD. Recall from your experience with the command-line tool that there was often a delay in discovering or registering services. You don't want to block the application while waiting for a reply to the function call. In addition, there may be more than one reply to a particular query. If you are browsing for services of a given type, there may be multiple instances of that type of service on the local network. Finally, you may wish to leave a service browser running so that you can track service instances as they come and go on the network.

The skeleton code in Example 7-1 provides an overview of the process. (As you will see later in this chapter, you follow each call to a core `DNSServiceDiscovery` function, such as `DNSServiceBrowse()`, with some code that enables the asynchronous callback.)

Example 7-1. Skeleton of `select()` loop

```
void HandleEvents(DNSServiceRef serviceRef)
{
    int dns_sd_fd = DNSServiceRefSockFD(serviceRef);
    // . . .
    while (!stopNow)
    {
        FD_ZERO(&readfds);
        FD_SET(dns_sd_fd, &readfds);
        // . . .
        tv.tv_sec = timeOut;
        tv.tv_usec = 0;
        int result = select(nfds, &readfds, (fd_set*)NULL, (fd_set*)NULL, &tv);
        if (result > 0)
        {
            // . . .
            if (FD_ISSET(dns_sd_fd, &readfds))
                err = DNSServiceProcessResult(serviceRef);
            if (err) stopNow = 1;
        }
    }
}
```

You can see the basic structure in the `HandleEvents()` example function shown in Example 7-1. First, you pass the initialized service discovery reference to `DNSServiceRefSockFD()`, to get the file descriptor for the Unix Domain Socket that's being used to communicate with the `mdnsd` daemon running in the background. The file descriptor is added to the set of file descriptors the process is watching. After the `select()` call returns, if the bit is set to indicate that the `DNSServiceDiscovery` file descriptor has data available for reading, then you call `DNSServiceProcessResult()`, which reads the message from the file descriptor, decodes it, and calls the appropriate callback function.

Note that a real networking program would probably be watching more file descriptors than just the DNS-SD one(s) and may also have time-based operations it's performing, too.

Example 7-2 shows a complete working example of a simple single-purpose `HandleEvents()` function. It runs until the user presses Ctrl-C to terminate the program, or one of the callback functions sets the `stopNow` variable. (If you're adding DNS-SD functionality to an existing `select()`-based application, you'd probably add the DNS-SD file descriptors to your existing `select()` loop rather than changing your existing code to use the example `select()` loop shown here.)

Example 7-2. HandleEvents using select()

```
#include <dns_sd.h>
#include <stdio.h>           // For stdout, stderr
#include <string.h>         // For strlen(), strcpy(), bzero()
#include <errno.h>         // For errno, EINTR
#include <time.h>

#ifdef _WIN32
#include <process.h>
typedef int pid_t;
#define getpid _getpid
#define strcasecmp _stricmp
#define sprintf _snprintf
#else
#include <sys/time.h>      // For struct timeval
#include <unistd.h>        // For getopt() and optind
#include <arpa/inet.h>     // For inet_addr()
#endif

// Note: the select() implementation on Windows (Winsock2)
//fails with any timeout much larger than this
#define LONG_TIME 10000000

static volatile int stopNow = 0;
static volatile int timeOut = LONG_TIME;

void
HandleEvents(DNSServiceRef serviceRef)
{
    int dns_sd_fd = DNSServiceRefSockFD(serviceRef);
    int nfds = dns_sd_fd + 1;
    fd_set readfds;
    struct timeval tv;
    int result;

    while (!stopNow)
    {
        FD_ZERO(&readfds);
        FD_SET(dns_sd_fd, &readfds);
        tv.tv_sec = timeOut;
        tv.tv_usec = 0;
```

Example 7-2. *HandleEvents using select() (continued)*

```
result = select(nfds, &readfds, (fd_set*)NULL, (fd_set*)NULL, &tv);
if (result > 0)
{
    DNSServiceErrorType err = kDNSServiceErr_NoError;
    if (FD_ISSET(dns_sd_fd, &readfds))
        err = DNSServiceProcessResult(serviceRef);
    if (err) stopNow = 1;
}
else
{
    printf("select() returned %d errno %d %s\n",
        result, errno, strerror(errno));
    if (errno != EINTR) stopNow = 1;
}
}
```

Accessing the Underlying Unix Domain Sockets

Your code interacts only indirectly with the Unix Domain Socket connection. `DNSServiceRefSockFD()` lets you get the raw file descriptor so you can add it to the set of file descriptors your `select()` loop is watching. When data arrives on this socket, you call `DNSServiceProcessResult()` to let the DNS-SD code decode the message and call your callback function. When you've finished what you're doing (say the user closes the browsing window), you call `DNSServiceRefDeallocate()` to stop the operation, close the Unix Domain Socket connection to the daemon, and free the resources and memory used to perform the operation.

DNSServiceRefSockFD()

In Example 7-2, the `DNSServiceRefSockFD()` function was called like this:

```
int dns_sd_fd = DNSServiceRefSockFD(serviceRef);
```

`dns_sd_fd` is given the value returned by `DNSServiceRefSockFD()`. This value is the underlying file descriptor of the service discovery reference specified by `serviceRef`. The signature of `DNSServiceRefSockFD()` is:

```
int DNSServiceRefSockFD (DNSServiceRef serviceRef);
```

DNSServiceProcessResult()

The function `DNSServiceProcessResult()` is used to call the appropriate callback function when there is a response from the `mdnsd` daemon running in the background. Its signature is deceptively simple:

```
DNSServiceErrorType DNSServiceProcessResult(DNSServiceRef serviceRef);
```

For proper use, start by referring again to Example 7-2, where a call to `DNSServiceProcessResult()` was made like this:

```
result = select(nfds, &readfds, (fd_set*)NULL, (fd_set*)NULL, &tv);
if (result > 0)
{
    DNSServiceErrorType err = kDNSServiceErr_NoError;
    if (FD_ISSET(dns_sd_fd, &readfds))
        err = DNSServiceProcessResult(serviceRef);
    if (err) stopNow = 1;
}
```

One important thing to note in this small snippet is that you only want to call `DNSServiceProcessResult()` if `select()`, or some similar system call, tells you data is ready. If you call `DNSServiceProcessResult()` when there is no data waiting to be read, it will block and wait until there is.

DNS Service Discovery error codes

Table 7-1 contains a listing of the possible error codes that arise when calling functions in the `DNSServiceDiscovery` APIs. Some errors, like `kDNSServiceErr_BadParam`, may be returned as an immediate result if you pass invalid parameters to a function. Others, like `kDNSServiceErr_NameConflict`, may be passed asynchronously to your callback function if an error condition occurs later.

The `DNSServiceDiscovery` error names are all of the form `kDNSServiceErr_NoErr`, `kDNSServiceErr_Unknown`, and so on. In Table 7-1, the initial `kDNSServiceErr_` part of each error name is omitted.

Table 7-1. Error codes

Error: <code>kDNSServiceErr_...</code>	Code	Description
<code>NoErr</code>	0	No error
<code>Unknown</code>	-65537	Unexpected error condition (should not happen)
<code>NoSuchName</code>	-65538	Given name does not exist
<code>NoMemory</code>	-65539	Out of memory (should not happen, except on devices with very limited memory)
<code>BadParam</code>	-65540	Parameter contains invalid data
<code>BadReference</code>	-65541	Reference being passed is invalid
<code>BadState</code>	-65542	Internal error (should not happen)
<code>BadFlags</code>	-65543	Invalid values for flags
<code>Unsupported</code>	-65544	Operation not supported
<code>NotInitialized</code>	-65545	Reference not initialized
<code>AlreadyRegistered</code>	-65547	Attempt to register a service that is registered
<code>NameConflict</code>	-65548	Attempt to register a service with an already used name

Table 7-1. Error codes (continued)

Error: kDNSServiceErr_...	Code	Description
Invalid	-65549	Certain invalid parameter data, such as domain name more than 255 bytes long
Incompatible	-65551	Client library incompatible with daemon (should never happen, unless installed daemon and client library are not the same version)
BadInterfaceIndex	-65552	Specified interface does not exist

DNSServiceRefDeallocate()

In Example 7-2, the `DNSServiceRef` was created elsewhere and passed into the `HandleEvents` routine. You get a `DNSServiceRef` when you call one of the core DNS-Service functions:

- `DNSServiceRegister()`
- `DNSServiceBrowse()`
- `DNSServiceResolve()`
- `DNSServiceEnumerateDomains()`
- `DNSServiceCreateConnection()`
- `DNSServiceQueryRecord()`

When this reference is no longer needed, it should be deallocated using:

```
void DNSServiceRefDeallocate(DNSServiceRef serviceRef);
```

When you call `DNSServiceRefDeallocate()`, the associated operation is stopped, the application's connection with the `mdnsd` daemon is terminated, the connecting socket is closed, and the memory associated with the reference is released. Before calling `DNSServiceRefDeallocate()`, make sure you've removed the socket from your `select()` loop, or you'll have a `select()` loop with a dead socket in it, which can cause confusing results, especially in multithreaded programs.

Using the DNSServiceDiscovery APIs

This section covers browsing for services, resolving services, registering services, and some of the less common operations, such as enumerating domains. In each case, there are three basic tasks to be performed: you initiate the operation, add the event source to your event loop, and provide the associated callback function. In the case of browsing, you call `DNSServiceBrowse()`. In the examples that use the code shown in Example 7-2, you will next call `HandleEvents()` and pass in the now initialized service discovery reference. Finally, you will implement a callback function, which will be called when the `mdnsd` daemon has a relevant reply. The form of the callback function is specified. So, for example, when browsing for services, the callback function must have the same signature as specified by the typedef `DNSServiceBrowseReply()`.

The number and type of the parameters passed to the callback function will be as indicated in the typedef. The process and the description of the parameters will quickly become familiar. After we finish describing browsing in detail, any repeated information will be summarized or omitted when describing resolving services, registering services, and enumerating domains.

Browsing for Services

To browse for available services, you need to call the `DNSServiceBrowse()` function and specify the type of service you are searching for and the domain in which to search. You also pass in the address of your callback function and the address of an uninitialized `DNSServiceRef`. After the `DNSServiceBrowse()` call has started the operation and initialized the `DNSServiceRef`, you can extract the underlying file descriptor to add it to your `select()` loop. Each time the `mdnsd` daemon responds, your `select()` loop will wake up, you call `DNSServiceProcessResult()`, and that calls your callback function for you. This section provides details for the `DNSServiceBrowse()` and `DNSServiceBrowseReply()` functions, along with a table listing possible flag values and an example of how you might browse for a specific type of service.

`DNSServiceBrowse()`

You initiate browsing for a service by calling `DNSServiceBrowse()`:

```
DNSServiceRef DNSServiceBrowse(
    DNSServiceRef *sdRef,
    DNSServiceFlags flags,
    uint32_t interfaceIndex,
    const char *regtype,
    const char *domain,
    DNSServiceBrowseReply callBack,
    void *context);
```

The first parameter is the address of your uninitialized service discovery reference.

The flags parameter is used for specifying optional settings that apply to some of the `DNSServiceDiscovery` routines. Currently, no optional settings are defined for the `DNSServiceBrowse()` call, so you should pass zero for this parameter.

Normally, applications pass 0 for `interfaceIndex`, and DNS-SD browses on all available interfaces. However, should you wish to restrict browsing to one specific interface, such as Ethernet or wireless, you can specify that interface by giving its interface index, as used in the `if_namestoindex()` family of functions. You can see each interface's index value by using the `ifconfig` command and looking for the `IPv6 scopeid` values. Interface indexes are typically small integers. For example, on Mac OS X machines, the Ethernet interface is often index 4, and AirPort is often index 5. The other novel value you can pass for this parameter is `kDNSServiceInterfaceIndexLocalOnly`. This restricts DNS-SD to only finding other services that were registered on the same machine (though the service itself is not

necessarily *running* on the same machine, because there can be proxies for services running on other machines). The applications that currently use this option are certain parallel processing products that have two versions at different prices—the single machine version and the network version. If the customer has only paid for the single machine version, the application only wants to find instances of the server process that are registered on the same machine, so it uses `kDNSServiceInterfaceIndexLocalOnly`.

The `regtype` is the same service type as entered when using the command-line tool in Chapter 6. It is the protocol followed by either `._tcp` or by `._udp`. For example, the `regtype` might have the value `_http._tcp`. The valid service names can be found at <http://www.dns-sd.org/ServiceTypes.html>.

The `domain` variable can have a specific value, such as `local`, or it can be `NULL` to indicate that the system should choose the appropriate list of domains to search. Generally, that list of domains to search will always include `local`, plus any additional unicast domains added explicitly by the user, plus any “legacy browse” domains automatically learned from the network.

The `callback` is the address of your callback function to be called when an instance of the specified service is found. Details on the callback function are contained in the next section. The callback function is also called in the event of asynchronous errors.

The `context` parameter is also passed to the callback function. This allows you to write a single callback function, which is used by several different browse operations, because the `context` parameter allows you to tell which particular `DNSServiceBrowse()` operation this event pertains to. Typically, the `context` parameter will be the address of some structure or object holding your state pertaining to that operation. C may not be an object-oriented language, but the `context` parameter here has an equivalent role to the “self” variable in an object-oriented language. You are free to use any value you wish for the `context` parameter, including `NULL`, and it will be passed unchanged to your callback function.

DNSServiceBrowseReply()

The callback function passed in as a parameter for the `DNSServiceBrowse()` function will have the following form:

```
void MyBrowseReply
(
    DNSServiceRef sdRef,
    DNSServiceFlags flags,
    uint32_t interfaceIndex,
    DNSServiceErrorType errorCode,
    const char *serviceName,
    const char *regtype,
    const char *replyDomain,
    void *context
);
```

The most interesting parameters here are `errorCode`, `serviceName`, `regtype`, and `replyDomain`. If `errorCode` is nonzero, then an error has occurred, as listed in Table 7-1. If `errorCode` is zero (`kDNSServiceErr_NoError`), then `serviceName`, `regtype`, and `replyDomain` tell you the name, type, and domain for the newly discovered (or removed) service.

The `sdRef` is the `DNSServiceRef` of the operation to which this callback relates. If you have multiple browse operations running at once, being handled by the same callback function, the callback function can use the `sdRef`, or the context parameter, to help it locate whatever private internal state is necessary for it to make sense of the result.

The `flags` parameter tells the callback function two interesting things. First, if the `kDNSServiceFlagsAdd` bit is set, then a new service has been discovered. If this bit is not set, then the named service, previously discovered, has gone away and should be removed from your onscreen display. Second, if the `kDNSServiceFlagsMoreComing` bit is set, the callback function should not bother updating its UI and repainting the screen right away, because more results are coming immediately after this one. Suppose you discover 100 service instances on the network—adding each one to the onscreen list individually and redrawing the window for every one will make a slow and flickery display. If, instead, you wait until all 100 are in your list in memory before updating the screen, then the entire service list appears virtually instantaneously, fully formed on the screen, instead of building up one line at a time.

Note that if the `kDNSServiceFlagsMoreComing` bit is not set, that does not mean that there are no more answers coming *ever*. What it means is that there are no more answers coming *right now*, so you should go ahead and update the screen display and perform any other relevant processing you may have deferred. Even after you get a callback with the `kDNSServiceFlagsMoreComing` bit not set, you could easily get another one just a millisecond later giving data newly discovered from the network, and, in fact, you should expect this to happen quite frequently. Don't make the mistake of canceling your browse operation because you got a callback with the `kDNSServiceFlagsMoreComing` bit not set, and you thought that meant that was the last answer you'd ever get.

The `interfaceIndex` tells you on which interface the service was discovered, particularly useful if you passed 0 for the `interfaceIndex` when calling `DNSServiceBrowse()`. Note that if your machine has both Ethernet and wireless, and there's some other machine connected via both Ethernet and wireless, then you will discover that machine's services twice, once via the Ethernet interface and once via the wireless interface. If one interface is turned off or disconnected, then you'll get remove events for only the service(s) discovered on the interface that went away. You therefore need to keep track of the interface indexes along with the name, type, and domain of each discovered service, so that when you receive remove events, you know which one to

remove. If you're particularly ambitious, you could also make your UI display include an icon indicating on which interface each service was discovered.

DNSServiceDiscovery flags

Table 7-2 contains a list of flags that are currently available to be used in the DNS-ServiceDiscovery APIs. The flags follow the format `kDNSServiceFlagsxxx`, where `xxx` represents one of the flags listed in the table. For the sake of brevity in the table, the `kDNSServiceFlags` portion is not written.

Table 7-2. *DNSServiceDiscovery flags*

Name <code>kDNSServiceFlags...</code>	Value	Description
<code>MoreComing</code>	1	Don't update UI; the callback will be called immediately with more results. If this bit is not set, it's time to update your UI now.
<code>Add</code>	2	Add service, domain, or record to list. If this bit is not set, then remove the service, domain, or record.
<code>Default</code>	4	Enumerated domain is default domain.
<code>NoAutoRename</code>	8	Prevents auto-renaming in case of name conflict.
<code>Shared</code>	16	Allows multiple records with same full domain name.
<code>Unique</code>	32	Specifies resource record name must be unique on the network.
<code>BrowseDomains</code>	64	Domains recommended for browsing are to be enumerated.
<code>RegistrationDomains</code>	128	Domains recommended for registration are to be enumerated.
<code>LongLivedQuery</code>	256	When using <code>DNSServiceQueryRecord</code> with unicast names, tells daemon to set up long-lived query with the server.

Browsing example

The example will have two functions. You will start browsing using the function `MyDNSServiceBrowse()`. In it, you declare a `DNSServiceErrorType` and `DNSServiceRef` variables. You then call `DNSServiceBrowse()` and specify that you are browsing for services of type `_http._tcp` on the local network and pass in the function named `MyBrowseCallBack` as the callback function. Pass the service discovery reference to the `HandleEvents()` function in Example 7-2. If there is an error reported, then use `DNSServiceRefDeallocate()` to clean up. In summary, `MyDNSServiceBrowse()` has the following outline:

```
static DNSServiceErrorType MyDNSServiceBrowse()
{
    DNSServiceErrorType error;
    DNSServiceRef serviceRef;
    error = DNSServiceBrowse(&serviceRef, /* parameters as described above */);
    if (!error)
    {
        HandleEvents(serviceRef);
    }
}
```

```

        DNSServiceRefDeallocate(serviceRef);
    }
    return error;
}

```

The second function is the callback function `MyBrowseCallback()`. If no error is reported, the flags are checked to see if the `kDNSServiceFlagsMoreComing` and `kDNSServiceFlagsAdd` flags are set. In this example code, a message is then printed to the screen that indicates whether the service is being added or removed, along with the service's name, type, and domain. If the `kDNSServiceFlagsMoreComing` flag is not set, then standard out is flushed to ensure that the information appears promptly on the user's screen. Both functions, along with `main()`, are in Example 7-3.

Example 7-3. DNSServiceBrowse example

```

#include <dns_sd.h>
#include <stdio.h>    // For stdout, stderr
#include <string.h>   // For strlen(), strcpy(), bzero()

extern void HandleEvents(DNSServiceRef);

static void
MyBrowseCallback(DNSServiceRef service,
                 DNSServiceFlags flags,
                 uint32_t interfaceIndex,
                 DNSServiceErrorType errorCode,
                 const char * name,
                 const char * type,
                 const char * domain,
                 void * context)
{
    #pragma unused(context)
    if (errorCode != kDNSServiceErr_NoError)
        fprintf(stderr, "MyBrowseCallback returned %d\n", errorCode);
    else
    {
        char *addString = (flags & kDNSServiceFlagsAdd) ? "ADD" : "REMOVE";
        char *moreString = (flags & kDNSServiceFlagsMoreComing) ? "MORE" : " ";
        printf("%-7s%-5s %d%s.%s%s\n",
              addString, moreString, interfaceIndex, name, type, domain);
    }
    if (!(flags & kDNSServiceFlagsMoreComing)) fflush(stdout);
}

static DNSServiceErrorType
MyDNSServiceBrowse()
{
    DNSServiceErrorType error;
    DNSServiceRef serviceRef;
}

```

Example 7-3. *DNSServiceBrowse* example (continued)

```
error = DNSServiceBrowse(&serviceRef,
                        0,                // no flags
                        0,                // all network interfaces
                        "_http._tcp",    // service type
                        "",              // default domains
                        MyBrowseCallBack, // call back function
                        NULL);           // no context
if (error == kDNSServiceErr_NoError)
{
    HandleEvents(serviceRef); // Add service to runloop to get callbacks
    DNSServiceRefDeallocate(serviceRef);
}

return error;
}

int main (int argc, const char * argv[])
{
    DNSServiceErrorType error = MyDNSServiceBrowse();
    if (error) fprintf(stderr, "DNSServiceDiscovery returned %d\n", error);
    return 0;
}
```

Save the code in Example 7-3 as *MyDNSSDBrowser.c* and the code in Example 7-2 as *DNSServiceCallbackSelect.c*, then compile and run them. If there are no services of type `_http._tcp` running on your local network, you can always follow the instructions in Chapter 6 to register a pretend one using the command-line tool. You can also use the example included in the section “Registering a Service,” later in this chapter.

Resolving a Service

The pattern for resolving is identical to that for browsing. To resolve, you call `DNSServiceResolve()` and then add the event source to your `select()` loop. When a result or results become available, your callback function will be called. This section details both functions and provides an example of resolving a registered service to determine its host and port.

DNSServiceResolve

To resolve a service, call `DNSServiceResolve()`:

```
DNSServiceErrorType DNSServiceResolve(
    DNSServiceRef *sdRef,
    DNSServiceFlags flags,
    uint32_t interfaceIndex,
    const char *name,
    const char *regtype,
```

```
const char *domain,  
DNSServiceResolveReply callback,  
void *context);
```

Pass `name`, `regtype`, and `domain` exactly as you received them in the `DNSServiceBrowse()` callback. If you are resolving a service that you discovered in a still-active browse call, then pass the discovered `interfaceIndex` to ensure that you resolve it on the specific interface on which it was discovered. If you are resolving a service that you discovered some time ago (perhaps saving its name, `regtype`, and `domain` in a preference file on disk), then you should set `interfaceIndex` to zero, because that service may now be available via a different interface. For example, the user could have originally discovered the service via Ethernet but now wants to use that same service via wireless.

When the daemon has the service information for you, it will call your callback function.

DNSServiceResolveReply

Your `DNSServiceResolve()` callback function needs to have the following form:

```
void MyDNSServiceResolveReply  
(  
    DNSServiceRef sdRef,  
    DNSServiceFlags flags,  
    uint32_t interfaceIndex,  
    DNSServiceErrorType errorCode,  
    const char *fullname,  
    const char *hosttarget,  
    uint16_t port,  
    uint16_t txtLen,  
    const char *txtRecord,  
    void *context  
);
```

As before, `errorCode` tells you if the operation was successful, and `sdRef` and `context` are provided to help your callback easily locate any state it needs. If you resolved without specifying a particular interface, then `interfaceIndex` tells you on which interface the answer was found.

The parameters `hosttarget` and `port` tell you where the service can be reached, today. Note that this can change over time. A given named service can be moved to a different machine. This is why it is important to store only the name, type, and domain in preference files on disk and resolve on demand when needed. If you store the host-name, port number, or even worse, the IP address, these could all be out of date when the user comes to access the service at a later date. The port is given in network byte order, exactly as needed for use in the `sin_port` field of a `struct sockaddr_in` you'd pass to `connect()`.

The `fullname` parameter gives you the fully qualified DNS name for the service, with all necessary escaping of dots, spaces, backslashes, and nonprinting characters,

making the name safe to pass to `DNSServiceQueryRecord()` or the standard Unix function `res_query()`. Most applications will never need to use this, but the `fullname` parameter is provided as a convenience for those that do. One example is `iChat`, which stores the user's picture as another DNS record with the same name as the service (SRV) record. Providing the properly escaped, fully qualified DNS name makes it easy for `iChat` to retrieve the image record using either `res_query()` or `DNSServiceQueryRecord()`.

The `txtLen` and `txtRecord` parameters give you additional optional information about the service, if present. The data is presented in raw DNS TXT record format. To help you decode this format, the `dns_sd.h` header file provides helper functions `TXTRecordContainsKey()` and `TXTRecordGetValuePtr()`, available in Mac OS X 10.4 and, later, Bonjour for Windows, Bonjour for Linux, etc.

```
int DNSSD_API TXTRecordContainsKey(uint16_t txtLen, const void *txtRecord,
    const char *key);

const void * DNSSD_API TXTRecordGetValuePtr(uint16_t txtLen, const void *txtRecord,
    const char *key, uint8_t *valueLen );
```

`TXTRecordContainsKey()` returns a Boolean `true/false` result indicating whether the named key appears in the text record.

`TXTRecordGetValuePtr()` returns a pointer to the value data for the named key. If the returned pointer is `NULL`, the named key did not appear in the text record or appeared with no `=` to indicate an associated value. If the returned pointer is non-`NULL`, then the named key appeared with an `=`. The `valueLen` indicates the length of the value data, which may be zero in the case of an empty value (i.e., "key=").

Resolution example

In Example 7-4, the service with the name `Not a real page` of type `_http._tcp` is resolved using `DNSServiceResolve()`. When `MyResolveCallback()` is called, if there has been no error, then the name of the service being resolved is displayed along with the hostname and port number.

Example 7-4. DNSServiceResolve Example

```
#include <dns_sd.h>
#include <stdio.h>           // For stdout, stderr
#include <string.h>         // For strlen(), strcpy(), bzero()

extern void HandleEvents(DNSServiceRef);

static void
MyResolveCallback(DNSServiceRef serviceRef,
    DNSServiceFlags flags,
    uint32_t interface,
    DNSServiceErrorType errorCode,
    const char *fullname,
```

Example 7-4. *DNSServiceResolve Example (continued)*

```
        const char *hosttarget,
        uint16_t port,
        uint16_t txtLen,
        const char *txtRecord,
        void *context)
{
    #pragma unused(flags)
    #pragma unused(fullname)

    if (errorCode != kDNSServiceErr_NoError)
        fprintf(stderr, "MyResolveCallBack returned %d\n", errorCode);
    else
        printf("RESOLVE: %s is at %s:%d\n", fullname, hosttarget, ntohs(port));
    if (!(flags & kDNSServiceFlagsMoreComing)) fflush(stdout);
}

static DNSServiceErrorType
MyDNSServiceResolve()
{
    DNSServiceErrorType error;
    DNSServiceRef serviceRef;

    error = DNSServiceResolve(&serviceRef,
                              0, // no flags
                              0, // all network interfaces
                              "Not a real page", //name
                              "_http_tcp", // service type
                              "local", //domain
                              MyResolveCallBack,
                              NULL); // no context

    if (error == kDNSServiceErr_NoError)
    {
        HandleEvents(serviceRef); // Add service to runloop to get callbacks
        DNSServiceRefDeallocate(serviceRef);
    }
    return error;
}

int
main (int argc, const char * argv[])
{
    DNSServiceErrorType error = MyDNSServiceResolve();
    fprintf(stderr, "DNSServiceDiscovery returned %d\n", error);
    //if function returns print error

    return 0;
}
```

Run this example by compiling it and running it with *DNSServiceCallbackSelect.c*. If you have a real `_http_tcp` service available, you can substitute its name in place of `Not a real page`. Alternatively, you can follow the instructions in Chapter 6 to register

a pretend `_http._tcp` service called `Not a real page` using the command-line tool, or you can proceed to the example in the section “Registering a Service” to write code to register your own pretend service with that name.

When you successfully resolve, you should see something like this:

```
RESOLVE: Not\032a\032real\032page._http._tcp.local is at SuiMai.local.:9092
```

Notice how the spaces in the fully qualified DNS name are replaced with `\032` as part of the escaping process to make the name safe to use with Unix routines such as `res_query()`.

Registering a Service

To register a service, you call the `DNSServiceRegister()` function and specify the name and the type of the service you are registering, as well as details about the interface, host, domain, and port. As with browsing, you pass in the address of the callback function, a context pointer, and an uninitialized `DNSServiceRef`, and then add the event source to your `select()` loop. Each time the `mdnsd` daemon responds, you call `DNSServiceProcessResult` and the callback function will be invoked. This section provides details for the `DNSServiceRegister()` and `DNSServiceRegisterReply()` functions, along with details of how to add, update, or remove a resource record using `DNSServiceAddRecord()`, `DNSServiceUpdateRecord()`, and `DNSServiceRemoveRecord()`. The section concludes with an example of how you might register a service.

DNSServiceRegister

To register a service, call the `DNSServiceRegister()` function:

```
DNSServiceErrorType DNSServiceRegister(  
    DNSServiceRef *sdRef,  
    DNSServiceFlags flags,  
    uint32_t interfaceIndex,  
    const char *name,  
    const char *regtype,  
    const char *domain,  
    const char *host,  
    uint16_t port,  
    uint16_t txtLen,  
    const void *txtRecord,  
    DNSServiceRegisterReply callBack,  
    void *context);
```

The parameters should seem familiar from the discussion of resolving services using `DNSServiceResolve()`. The difference is that now, instead of learning the host, port, and DNS TXT record data, you’re providing it.

The `interfaceIndex` parameter allows you to advertise your service on only one specific interface, if desired. Most applications use zero for `interfaceIndex`. The other novel value you can pass for this parameter is `kDNSServiceInterfaceIndexLocalOnly`,

which means that your service will not actually be advertised on the network but only made visible to other browsing clients on the same machine. One use of this is for background processes that provide a web-based user interface for configuration but (perhaps for security reasons) only allow configuration from the local machine, not remotely over the network. By advertising their service using `kDNSServiceInterfaceIndexLocalOnly`, their configuration page will appear in the Bonjour list in web browsers running on the local machine only, not on other machines on the network.

The `name` parameter is optional. If you pass `NULL` or an empty string, a system-wide default name is used for your service. For many services, when there is only usually one instance of that service on a given machine, using a system-wide default name is a sensible choice. Whether you specify an explicit name or use the system-wide default, Multicast DNS will ensure that it is unique on the local network. For example, if you advertise an HTTP server with the name “Web Server” and there is already a different HTTP service on the network with the same name, then yours will be automatically renamed to “Web Server (2).” If there’s already a “Web Server (2),” then yours will be automatically renamed to “Web Server (3),” and so on. If you don’t want this auto-rename behavior, then use the flag `kDNSServiceFlagsNoAutoRename`, and instead of renaming automatically, DNS-SD will call your callback function with a `kDNSServiceErr_NameConflict` error result so you can pick a new name for yourself. When this happens, your service registration will have been terminated. You will need to remove it from your `select()` loop, destroy the `DNSServiceRef` using `DNSServiceRefDeallocate()`, and then try again with a new name.

The `domain` parameter is optional. If you pass `NULL` or an empty string, it means “pick a sensible default for me,” which is what most applications do. Usually the sensible default will be `local`, possibly plus one wide-area domain as selected by the user.

The `hostname` parameter is optional. If you pass `NULL` or an empty string, it means the current host, which is what most applications do. You only need to specify a hostname when creating proxy registrations for services running on some other machine.

The port number is in network byte order. If you’re using a fixed port number, then this is exactly as you would have used in the `sin_port` field of your `struct sockaddr_in` in your `bind()` call. If you’re using a dynamic port number, this is exactly as you would have received it in the `sin_port` field of your `struct sockaddr_in` in your `getsockname()` call.

Some service types have extra data stored in the TXT record. For example, with HTTP services, the path to the page in question can be stored in the TXT record, e.g., `path=/index.html`. You need to provide a pointer to a properly formatted DNS TXT record. To help in the creation of properly formatted DNS TXT records, you can use the `TXTRecordCreate()` family of helper functions from the `dns_sd.h` header file, available in Mac OS X 10.4 and, later, Bonjour for Windows, Bonjour for Linux, etc.

DNSServiceRegisterReply

The following is the prototype for the callback function used when registering a service:

```
void MyRegisterReply
(
    DNSServiceRef sdRef,
    DNSServiceFlags flags,
    DNSServiceErrorType errorCode,
    const char *name,
    const char *regtype,
    const char *domain,
    void *context
);
```

The parameters are similar to those for `DNSServiceBrowseReply()`. Of particular interest is the name parameter. When auto-renaming is in effect, the name parameter tells you what name Multicast DNS finally picked for you. In some kinds of application, knowing your own name is important. For example, iChat kind of acts as both a client and a server on the same machine. When the client side of iChat browses, it finds all service instances, including its own. By knowing its own name, it can filter itself out of the list it presents to the user.

DNSServiceAddRecord

Having registered a service, `DNSServiceAddRecord()` lets you add additional records with the same name as the service (SRV) record. This is rare, and most applications never need to do this. One example of an application that uses this is iChat. iChat uses this to add a separate record containing the user's picture. When you add a record using `DNSServiceAddRecord()`, you get a `DNSRecordRef`, which you can subsequently use in `DNSServiceUpdateRecord()` to update the record's data and in `DNSServiceRemoveRecord()` to remove the record from the service registration.

Here is the signature of `DNSServiceAddRecord()`:

```
int DNSServiceAddRecord(
    DNSServiceRef sdRef,
    DNSRecordRef *RecordRef,
    DNSServiceFlags *flags,
    uint16_t rrtype,
    uint16_t rdlen,
    const void *rdata,
    uint32_t ttl
);
```

The `sdRef` parameter specifies the service you're adding the record to, and the specified `DNSRecordRef` is initialized as a result of this call.

The last four parameters are used to describe the record being added. The `rrtype` is the numerical value of the resource record type. The list of these numbers can be found at <http://www.iana.org/assignments/dns-parameters> and in the `dns_sd.h` header file. The `rdlen` is the length in bytes of `rdata`, which is (in theory) up to 64 KB of

opaque binary data that is to be stored in the resource record being added. In practice, 100 bytes or 200 bytes is a reasonable size, and anything above 1,000 bytes can be inefficient on the network. Finally, `ttl` is a 32-bit signed value indicating the record's requested time to live in seconds.

DNSServiceUpdateRecord

Use `DNSServiceUpdateRecord()` to request an update to a DNS record.

```
DNSServiceErrorType DNSServiceUpdateRecord(
    DNSServiceRef sdRef,
    DNSRecordRef recordRef,
    DNSServiceFlags flags,
    uint16_t rdlen,
    const void *rdata,
    uint32_t ttl
);
```

`RecordRef` identifies the record to be updated. Either it is a `DNSRecordRef` created by `DNSServiceAddRecord()`, `DNSServiceRegisterRecord()`, or `NULL`, which means “Update the service's primary TXT record.” Most services never update their TXT records. Again, the exception is iChat, which uses its TXT record to show the user's available/idle/away state, and consequently updates it all the time.

The `rdlen`, `rdata`, and `ttl` have the same meanings that were described for `DNSServiceAddRecord()`.

DNSServiceRemoveRecord

To request the removal of a resource record from a service's registration information, call `DNSServiceRemoveRecord()`.

```
DNSServiceErrorType DNSServiceRemoveRecord(
    DNSServiceRef sdRef,
    DNSRecordRef recordRef,
    DNSServiceFlags flags
);
```

Registration example

The registration example shown in Example 7-5 is similar to the browse example shown in Example 7-3. The function `MyDNSServiceRegister()` has roughly the same format as `MyDNSServiceBrowse()`. The callback function, `MyRegisterCallback()`, reports that the service has been registered if no errors are reported and reports the errors if any exist. Here is the entire listing.

Example 7-5. DNSServiceRegister example

```
#include <dns_sd.h>
#include <stdio.h>           // For stdout, stderr
#include <string.h>         // For strlen(), strcpy(), bzero()
```

Example 7-5. DNSServiceRegister example (continued)

```
extern void HandleEvents(DNSServiceRef);

static void
MyRegisterCallBack(DNSServiceRef service,
                  DNSServiceFlags flags,
                  DNSServiceErrorType errorCode,
                  const char * name,
                  const char * type,
                  const char * domain,
                  void * context)
{
    #pragma unused(flags)
    #pragma unused(context)

    if (errorCode != kDNSServiceErr_NoError)
        fprintf(stderr, "MyRegisterCallBack returned %d\n", errorCode);
    else
        printf("%-15s %s.%s%s\n", "REGISTER", name, type, domain);
}

static DNSServiceErrorType MyDNSServiceRegister()
{
    DNSServiceErrorType error;
    DNSServiceRef serviceRef;

    error = DNSServiceRegister(&serviceRef,
                              0, // no flags
                              0, // all network interfaces
                              "Not a real page", // name
                              "_http._tcp", // service type
                              "", // register in default domain(s)
                              NULL, // use default host name
                              htons(9092), // port number
                              0, // length of TXT record
                              NULL, // no TXT record
                              MyRegisterCallBack, // call back function
                              NULL); // no context

    if (error == kDNSServiceErr_NoError)
    {
        HandleEvents(serviceRef);
        DNSServiceRefDeallocate(serviceRef);
    }

    return error;
}

int main (int argc, const char * argv[])
{
    DNSServiceErrorType error = MyDNSServiceRegister();
    fprintf(stderr, "DNSServiceDiscovery returned %d\n", error);
    return 0;
}
```

Save the code in Example 7-5 as *MyDNSSDRegistrar.c* and compile it and run it along with *DNSServiceCallbackSelect.c*. Note that the port number for this service has been hardcoded to 9092. When you run this example, you should see the following message:

```
REGISTER      Not a real page._http._tcp.local.
```

You can verify that this service has been registered using the *dns-sd* command-line tool or by running the previous example of browsing. You should see this message:

```
ADD          Not a real page._http._tcp.local.
```

Enumerating Domains

To date, most of the use of DNS-SD has been for link-local multicast discovery. It is natural that this is the area that would get the most interest, because this was the area of IP most desperately in need of improvement.

However, as link-local DNS-SD becomes mature, people begin to look outward to wide-area service discovery. When you begin to browse domains other than *local*, the question arises, “How does the machine know which domains to browse?” Forcing the user to configure this manually would not be in keeping with the spirit of Zero Configuration Networking. DNS-SD has mechanisms to learn this information from the local network, and the domain enumeration functions allow applications to access this information to present a good user interface. As with the other DNS-SD APIs, these interfaces are asynchronous and ongoing until cancelled, because information from the network can change at any time.

When an application wants to enable browsing in multiple domains, it asks for the list of recommended browsing domains. It should not automatically browse every domain it finds, because that would be extremely expensive on the network. Instead, it should present the list of domains to the user so he can pick. One domain will be delivered with the `kDNSServiceFlagsDefault` flag set, and that domain should be highlighted by default in the browser. It’s the network equivalent of the “you are here” marker on a map. The list is purely advisory; users should still be allowed to manually enter additional domains to browse if they wish.

When an application wants to register its service in domains other than just *local*, it can ask for the list of recommended registration domains. As before, an application should not automatically register in every domain it finds. The list is intended to be shown to the user, so the user may pick one domain from the list. Also as before, the list is advisory, meaning users should be allowed to manually enter a different domain to register in if they wish. Most applications will not need to enumerate registration domains, because they will simply use the user’s configured system-wide default by passing `NULL` for the domain when registering.

To enumerate domains, you need to call `DNSServiceEnumerateDomain()` and use a callback function built on the template provided by `DNSServiceDomainEnumReply` to

collect the information. If the network administrator has not created the domain enumeration records described in Chapter 5, the only result you will get is *local*.

In this section, the signatures for `DNSServiceEnumerateDomain()` and `DNSServiceDomainEnumReply()` are provided along with an example of using this part of the API.

DNSServiceEnumerateDomain

Begin searching for recommended browsing or registration domains using `DNSServiceEnumerateDomain()`:

```
DNSServiceErrorType DNSServiceEnumerateDomains(  
    DNSServiceRef *sdRef,  
    DNSServiceFlags flags,  
    uint32_t interfaceIndex,  
    DNSServiceDomainEnumReply callBack,  
    void *context);
```

Set the flag either using `kDNSServiceFlagsBrowseDomains` to return the domains recommended for browsing or using `kDNSServiceFlagsRegistrationDomains` to return the domains recommended for registering services. The `sdRef` is an uninitialized service discovery reference that will be initialized when `DNSServiceEnumerateDomain()` is called. The remaining variables are as they were for `DNSServiceBrowse()` and `DNSServiceRegister()`.

DNSServiceDomainEnumReply

The callback function for enumerating domains must be modeled on `DNSServiceDomainEnumReply()`.

```
void MyDNSServiceDomainEnumReply  
(  
    DNSServiceRef sdRef,  
    DNSServiceFlags flags,  
    uint32_t interfaceIndex,  
    DNSServiceErrorType errorCode,  
    const char *replyDomain,  
    void *context  
);
```

The variables have been described previously for `DNSServiceRegisterReply()` and `DNSServiceBrowseReply()`. Consult Table 7-2 for the flags that can be passed in. As before, `kDNSServiceFlagsMoreComing` indicates that you should wait to update your UI, as the callback function will be called again immediately. If `kDNSServiceFlagsAdd` is set, then the domain pointed to should be added to the list of domains, and if `kDNSServiceFlagsAdd` is not present, then the domain pointed to should be removed from the list. The `kDNSServiceFlagsDefault` flag is set if the domain is the domain that should be selected by default.

Enumeration example

The enumeration example shown in Example 7-6 follows the same pattern as the previous examples. The call to `DNSServiceEnumerateDomains()` passes in the flag for browse domains. In the callback function, the discovered domains are displayed, along with an indication of whether there are `kDNSServiceFlagsMoreComing` and with a field showing whether the domain is being added or removed from the list and whether it is a default domain.

Example 7-6. `DNSServiceEnumerateDomains()` example

```
#include <dns_sd.h>
#include <stdio.h>           // For stdout, stderr
#include <string.h>         // For strlen(), strcpy(), bzero()

extern void HandleEvents(DNSServiceRef);

static void
MyEnumerateBrowseDomainsCallBack(DNSServiceRef sdRef,
                                  DNSServiceFlags flags,
                                  uint32_t interface,
                                  DNSServiceErrorType errorCode,
                                  const char *replyDomain,
                                  void *context)
{
    #pragma unused(context)

    if (errorCode != kDNSServiceErr_NoError)
        fprintf(stderr, "EnumerateDomainsCallBack returned %d\n", errorCode);
    else
    {
        char *moreString = (flags & kDNSServiceFlagsMoreComing) ? "MORE" : "";
        char *addString = "REMOVE";
        if (flags & kDNSServiceFlagsAdd)
            addString = (flags & kDNSServiceFlagsDefault) ? "DEFAULT" : "ADD";
        printf("%-8s%-5s%s\n", addString, moreString, replyDomain);
    }

    if (!(flags & kDNSServiceFlagsMoreComing)) fflush(stdout);
}

static DNSServiceErrorType
MyDNSServiceEnumerateBrowse()
{
    DNSServiceErrorType error;
    DNSServiceRef serviceRef;

    error = DNSServiceEnumerateDomains(
        &serviceRef,
        kDNSServiceFlagsBrowseDomains, // browse domains
        0, // all network interfaces
        MyEnumerateBrowseDomainsCallBack, //callback function
        NULL); // no context
```

Example 7-6. *DNSServiceEnumerateDomains()* example (continued)

```
    if (error == kDNSServiceErr_NoError)
    {
        HandleEvents(serviceRef); // Add service to runloop to get callbacks
        DNSServiceRefDeallocate(serviceRef);
    }
    return error;
}

int
main (int argc, const char * argv[])
{
    DNSServiceErrorType error = MyDNSServiceEnumerateBrowse();
    if (error) fprintf(stderr, "DNSServiceDiscovery returned %d\n", error);
    return 0;
}
```

You can run this example by compiling it and running it with *DNSServiceCallback-Select.c*. As mentioned before, if your network administrator has not created any domain enumeration records, the only result you will get is local.

```
ADD          local.
```

Other Operations

This section outlines some of the other lesser-used functions from *dns_sd.h*.

DNSServiceCreateConnection() and *DNSServiceRegisterRecord()* are used by applications that need to create a large number of records. A single *DNSServiceRef* is created using *DNSServiceCreateConnection()*, and then multiple records are registered on that single connection.

DNSServiceQueryRecord() allows the client to query for any arbitrary DNS record, with any name, type, or class, unicast or multicast. It is, in many ways, similar to the standard Unix *res_query()* function, except that it operates asynchronously with a callback function.

DNSServiceReconfirmRecord() is used for cache management. Multicast DNS caches data for efficiency, but anytime data is cached, it can become out of date. If a client believes that data is out of date, it can call *DNSServiceReconfirmRecord()* to provide a hint to the cache management algorithm. For example, suppose a client gets a host's address record using *DNSServiceQueryRecord()*, but the host does not respond. If the client calls *DNSServiceReconfirmRecord()*, then Multicast DNS will requery for the record, and if no response is received, then the record will be deleted from the cache. In addition, any SRV records referencing that target host will automatically be considered potentially suspect and will, in turn, be reconfirmed. If the SRV records are not confirmed, then they too will be deleted from the cache, and any PTR records referencing the now-departed SRV records will also be considered potentially suspect. After a short time, if these records are not confirmed, they will also be deleted

from the cache. The end result of this is that services being advertised from the departed host will disappear from browsing lists soon, instead of waiting a full hour for the record's TTL to expire. This call is highly specialized, and most applications will never have to use it, because in normal cases cache reconfirmation is handled automatically. In normal cases, if a host crashes, then the `DNSServiceQueryRecord()` call to look up its IP address will fail, and that will automatically kick off the chain of reconfirmations that purges the stale SRV and PTR records, too.

Event Handling with Cocoa RunLoop or Core Foundation CFRunLoop

So far in this chapter, you have used the `select()` loop to receive the asynchronous event notifications central to DNS-SD. In this last section, you will see alternative event delivery solutions. First, without any changes to the code presented in Examples 7-3 through 7-6, you can swap out the cross-platform code presented in Example 7-2 for a Mac OS X specific run loop that uses Core Foundation classes. Second, you will see how to alter the code to take advantage of the Windows-specific event loop.

If you're writing a Cocoa or Core Foundation application, you'll probably be using a Cocoa RunLoop or Core Foundation CFRunLoop (which are actually the same thing under the covers). You'll want to add a Cocoa- or Core Foundation-compatible event-generating object to your RunLoop. To do that, you extract the Unix Domain Socket from the `DNSServiceRef`, construct a `CFSocket` from that, and then construct a `CFRunLoopSourceRef` from that. The rough outline is shown in Example 7-7.

Example 7-7. Skeleton of CFRunLoop

```
typedef struct MyDNSServiceState
{
    DNSServiceRef      service;
    CFSocketRef        socket;
    CFRunLoopSourceRef source;
} MyDNSServiceState;

void
HandleEvents(DNSServiceRef serviceRef)
{
    // . . .
    // Access the underlying Unix domain socket and create a CFSocket
    sock = DNSServiceRefSockFD(ref->service);
    ref->socket = CFSocketCreateWithNative(NULL, sock,
        kCFSocketReadCallBack, MySocketCallBack, &context);
    // . . .
    // Create a CFRunLoopSource from the CFSocket, add to run loop and start.
    ref->source = CFSocketCreateRunLoopSource(NULL, ref->socket, 0);
    CFRunLoopAddSource(CFRunLoopGetCurrent(), ref->source, kCFRunLoopCommonModes);
}
```

Example 7-7. Skeleton of *CFRunLoop* (continued)

```
// . . .

CFRunLoopRun();
}
```

`DNSServiceRefSockFD()` extracts the Unix Domain Socket from the `DNSServiceRef`.

`CFSocketCreateWithNative()` makes a `CFSocket` from a native Unix socket.

`CFSocketCreateRunLoopSource()` makes a `CFRunLoopSource` from the `CFSocket`.

Once added to the `RunLoop`, `MySocketCallback()` will be called every time there is data waiting to be read. You can use a single `MySocketCallback()` routine for all of your DNS-SD operations. All it has to do is call `DNSServiceProcessResult()` on the right `DNSServiceRef`, and DNS-SD will do the rest to invoke the right callbacks.

```
static void
MySocketCallback(CFSocketRef s, CFSocketCallbackType type,
                 CFDataRef address, const void * data, void * info)
{
    // . . . cast the context info to initialize ref
    MyDNSServiceState * ref = (MyDNSServiceState *)info;
    // . . . use the service discovery reference for callback
    err = DNSServiceProcessResult(ref->service);
    // handle error . . .
}
```

Example 7-8 shows the full listing with all of the details.

Example 7-8. Core Foundation *RunLoop* example

```
// Simple example of how to handle DNSServiceDiscovery events using a CFRunLoop
```

```
#include <dns_sd.h>
#include <CoreFoundation/CoreFoundation.h>
```

```
// Structure to hold CFRunLoop-related state
```

```
typedef struct MyDNSServiceState
{
    DNSServiceRef      service;    // Active DNSServiceDiscovery operation
    CFSocketRef        cfsocket;   // CFSocket for this operation
    CFRunLoopSourceRef source;     // RunLoopSource for this CFSocket
} MyDNSServiceState;
```

```
// Remove a DNSServiceDiscovery operation from a CFRunLoop's
// set of active event sources
```

```
static void DNSServiceRemoveSource(CFRunLoopRef rl, MyDNSServiceState *ref)
{
    assert(rl != NULL);
    assert(ref != NULL);
}
```

Example 7-8. Core Foundation RunLoop example (continued)

```
// Remove the CFRunLoopSource from the current run loop.
CFRunLoopRemoveSource(r1, ref->source, kCFRunLoopCommonModes);
CFRelease(ref->source);

// Invalidate the CFSocket.
CFSocketInvalidate(ref->cfsocket);
CFRelease(ref->cfsocket);

// Workaround to give time to CFSocket's select() thread
// so it can remove the socket from its FD set before we
// close the socket by calling DNSServiceRefDeallocate.
usleep(1000);

// Terminate the connection with the daemon, which cancels the operation.
DNSServiceRefDeallocate(ref->service);
free(ref);
}

// Helper function: When CFRunLoop indicates an interesting event,
// this function calls DNSServiceProcessResult() to handle it

static void MySocketCallBack(CFSocketRef s, CFSocketCallBackType type,
                           CFDataRef address, const void *data, void *info)
{
    #pragma unused(s)
    #pragma unused(type)
    #pragma unused(address)
    #pragma unused(data)

    DNSServiceErrorType err;
    MyDNSServiceState *ref = (MyDNSServiceState *)info;
    assert(ref != NULL);

    // Read a reply from the daemon, which will call the appropriate callback.
    err= DNSServiceProcessResult(ref->service);
    if (err != kDNSServiceErr_NoError)
    {
        fprintf(stderr, "DNSServiceProcessResult returned %d\n", err);
        // Terminate the discovery operation and release everything.
        DNSServiceRemoveSource(CFRunLoopGetCurrent(), ref);
    }
}

// Add a DNSServiceDiscovery operation to a CFRunLoop's
// set of active event sources

static void DNSServiceAddSource(CFRunLoopRef r1, MyDNSServiceState *ref)
{
    CFSocketContext context = { 0, ref, NULL, NULL, NULL };
    CFSocketNativeHandle sock = DNSServiceRefSockFD(ref->service);
    assert(sock != -1);
```

Example 7-8. Core Foundation RunLoop example (continued)

```
// Create a CFSocket from the underlying Unix Domain socket.
ref->cfsocket = CFSocketCreateWithNative(NULL, sock,
    kCFSocketReadCallBack, MySocketCallBack, &context);

// Prevent CFSocketInvalidate from closing DNSServiceRef's socket.
CFOptionFlags f = CFSocketGetSocketFlags(ref->cfsocket);
CFSocketSetSocketFlags(ref->cfsocket, f & ~kCFSocketCloseOnInvalidate);

// Create a CFRRunLoopSource from the CFSocket.
ref->source = CFSocketCreateRunLoopSource(NULL, ref->cfsocket, 0);

// Add the CFRRunLoopSource to the current run loop.
CFRunLoopAddSource(rl, ref->source, kCFRunLoopCommonModes);
}

// Simple example: Here we just add a single DNSServiceDiscovery event source,
// and then call CFRRunLoopRun() to handle the events. In a program that already
// has a main RunLoop, you'd just keep that as is, and use DNSServiceAddSource/
// DNSServiceRemoveSource to add and remove event sources from that RunLoop.

void HandleEvents(DNSServiceRef serviceRef)
{
    MyDNSServiceState ref = { serviceRef };
    DNSServiceAddSource(CFRRunLoopGetCurrent(), &ref);

    CFRRunLoopRun();
}
```

Save this as *DNSServiceCallbackCF.c*. Compile and run this with any of the files from Examples 7-3 to 7-6 in place of *DNSServiceCallbackSelect.c*. The results should be the same as before.

Note that because a Cocoa RunLoop and a Core Foundation CFRRunLoop are actually the same thing (Cocoa and Core Foundation just provide their own APIs to access the same underlying object), the code shown above can also be used in an Objective-C program. It may not look much like Objective-C (no square brackets all over the place), but that's no problem. The Objective-C compiler also fully supports standard C, and the code will compile and do exactly what you need.

DNSServiceDiscovery's context parameter helps us interface with Objective-C's object-oriented paradigm. When calling one of the DNSServiceDiscovery API routines, pass in the address of your C-style callback function and, for the context parameter, pass a reference to the object (usually *self*) that you want to handle the events, like this:

```
DNSServiceBrowse(&ref, 0, 0, srvtype, "", BrowseReplyFn, self);
```

Then, in your C-style callback function, you recover the object context by writing something like `MyObjectType *self = (MyObjectType *)context`, as shown here:

```
void BrowseReplyFn(DNSServiceRef sdRef, DNSServiceFlags flags,
                  uint32_t interfaceIndex, DNSServiceErrorType errorCode,
                  const char *serviceName, const char *regtype, const char *replyDomain,
                  void *context)
{
    MyObjectType *self = (MyObjectType *)context;
    [self doThis];
    [self doThat];
    [self addName: [NSString stringWithUTF8String:serviceName]];
    //... and so on
}
```

Event Handling with Microsoft Windows GetMessage() Message Loop

There are some differences in working with the Windows event loop. In this example, you will configure and create a window that is not displayed in the application. This gives you a template for developing GUI-based Zeroconf applications for Windows. The Zeroconf events are processed as messages to this window. Here is the outline of the `HandleEvents()` function for Windows:

```
void HandleEvents(DNSServiceRef inServiceRef)
{
    //... Configure and create a window that is not shown but that
    // is used to process DNS-SD events as messages to the window.

    wind = CreateWindow(wcex.lpszClassName, wcex.lpszClassName, 0,
                       CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL,
                       NULL, instance, NULL);

    // ... Associate the DNS-SD browser with our window

    err = WSAAsyncSelect((SOCKET) DNSServiceRefSockFD(gServiceRef), wind,
                        DNSSD_EVENT, FD_READ | FD_CLOSE);

    assert(err == kDNSServiceErr_NoError);

    // DNS-SD events are dispatched while in this loop.

    while(GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    // Clean up.
```

```

        WSASyncSelect((SOCKET) DNSServiceRefSockFD(gServiceRef), wind,
                    DNSSD_EVENT, 0);
    }

```

When a window event is received, it is processed in the callback function `WndProc()`. If the event is a DNS-SD event, it is passed on to `DNSServiceProcessResult()` as before. Here is the sketch of `WndProc()`:

```

static LRESULT CALLBACK WndProc(HWND inWindow, UINT inMsg,
                                WPARAM inWParam, LPARAM inLParam)
{
    LRESULT result;
    switch(inMsg)
    {
        case DNSSD_EVENT:
            DNSServiceProcessResult(gServiceRef);
            result = 0;
            break; //...
    }
    return(result);
}

```

All of the details are provided in the code listing in Example 7-9.

Example 7-9. Windows event loop example

```

#include    "stdafx.h"
#include    <assert.h>
#include    <stdio.h>
#include    <dns_sd.h>

// Constants

#define DNSSD_EVENT        (WM_USER + 0x100)
                        // Message sent to Window when a DNS-SD event occurs.

// Prototypes

void HandleEvents(DNSServiceRef inServiceRef);

static LRESULT CALLBACK WndProc(HWND inWindow,
                                UINT inMsg,
                                WPARAM inWParam,
                                LPARAM inLParam);

static void DNSSD_API
    BrowserCallBack(DNSServiceRef    inServiceRef,
                   DNSServiceFlags inFlags,
                   uint32_t inIFI,
                   DNSServiceErrorType inError,
                   const char * inName,
                   const char * inType,
                   const char * inDomain,
                   void * inContext);

```

Example 7-9. Windows event loop example (continued)

```
// Globals

static DNSServiceRef      gServiceRef = NULL;

// Main entry point for application.

void HandleEvents(DNSServiceRef inServiceRef)

{
    HINSTANCE      instance;
    WNDCLASSEX     wcex;
    HWND           wind;
    MSG            msg;
    int            err;

    gServiceRef = inServiceRef;

    // Create the window. This window won't actually be shown,
    // but it demonstrates how to use DNS-SD with Windows GUI
    // applications by having DNS-SD events processed as messages
    // to a Window.

    instance = GetModuleHandle(NULL);
    assert(instance);

    wcex.cbSize = sizeof(wcex);
    wcex.style = 0;
    wcex.lpfnWndProc = (WNDPROC) WndProc;
    wcex.cbClsExtra = 0;
    wcex.cbWndExtra = 0;
    wcex.hInstance = instance;
    wcex.hIcon = NULL;
    wcex.hCursor = NULL;
    wcex.hbrBackground = NULL;
    wcex.lpszMenuName = NULL;
    wcex.lpszClassName = TEXT("ZeroconfExample");
    wcex.hIconSm = NULL;
    RegisterClassEx(&wcex);

    wind = CreateWindow(wcex.lpszClassName, wcex.lpszClassName, 0,
                       CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL,
                       NULL, instance, NULL);

    assert(wind);

    // Associate the DNS-SD browser with our window
    // using the WSAASyncSelect mechanism. Whenever something
    // related to the DNS-SD browser occurs, our private Windows message
    // will be sent to our window so we can give DNS-SD a
    // chance to process it. This allows DNS-SD to avoid using a
    // secondary thread (and all the issues with synchronization that
    // would introduce), but still process everything asynchronously.
```

Example 7-9. Windows event loop example (continued)

```
// This also simplifies app code because DNS-SD will only run when we
// explicitly call it.

err = WSAAsyncSelect((SOCKET) DNSServiceRefSockFD(gServiceRef), wind,
                    DNSSD_EVENT, FD_READ | FD_CLOSE);

assert(err == kDNSServiceErr_NoError);

// Main event loop for the application. All DNS-SD events are
// dispatched while in this loop.

while(GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}

// Clean up. This is not strictly necessary since the normal
// process cleanup will close DNS-SD socket(s) and release memory,
// but it's here to demonstrate how to do it.

WSAAsyncSelect((SOCKET) DNSServiceRefSockFD(gServiceRef), wind,
              DNSSD_EVENT, 0);
}

// Callback for the Window. DNS-SD events are delivered here.

static LRESULT CALLBACK WndProc(HWND inWindow, UINT inMsg,
                               WPARAM inWParam, LPARAM inLParam)
{
    LRESULT result;

    switch(inMsg)
    {
        case DNSSD_EVENT:

            // Process the DNS-SD event. All DNS-SD callbacks occur from
            // within this function.

            if (DNSServiceProcessResult(gServiceRef) != kDNSServiceErr_NoError)
                result = -1;
            else
                result = 0;
            break;

        default:
            result = DefWindowProc(inWindow, inMsg, inWParam, inLParam);
            break;
    }

    return(result);
}
```

Event Handling with Microsoft Windows MFC

If you're programming using the Microsoft Windows MFC (Microsoft Foundation Classes) programming model, then you don't need your own event loop. You just declare the messages that your window accepts, and, as events happen, MFC sends the appropriate message to your window object. Example 7-10 shows an outline of what you need to do to integrate (in this example) DNS-SD browsing into an MFC application.

Example 7-10. Windows MFC example

```
#include "stdafx.h"
#include <dns_sd.h>
#include <winsock2.h>

#define WM_PRIVATE_SERVICE_EVENT          ( WM_USER + 0x100 )

class MyWindow
:
    public CWnd
{
public:

    MyWindow();
    virtual ~MyWindow( void );

protected:

    // General

    afx_msg int    OnCreate( LPCREATESTRUCT inCreateStruct );
    afx_msg void    OnDestroy( void );
    afx_msg LONG    OnServiceEvent( WPARAM inWParam, LPARAM inLParam );

    // Browsing

    static void DNSSD_API
    BrowseReply(
        DNSServiceRef        inRef,
        DNSServiceFlags      inFlags,
        uint32_t             inInterfaceIndex,
        DNSServiceErrorType  inErrorCode,
        const char *         inName,
        const char *         inType,
        const char *         inDomain,
        void *               inContext );

    DECLARE_MESSAGE_MAP()

private:

    DNSServiceRef m_serviceRef;
};
```

Example 7-10. Windows MFC example (continued)

```
BEGIN_MESSAGE_MAP( MyWindow, CWnd )
    ON_WM_CREATE( )
    ON_WM_DESTROY( )
    ON_MESSAGE( WM_PRIVATE_SERVICE_EVENT, OnServiceEvent )
END_MESSAGE_MAP( )

int
MyWindow::OnCreate( LPCREATESTRUCT inCreateStruct )
{
    DNSServiceErrorType err;

    err = CWnd::OnCreate( inCreateStruct );

    if ( err )
        goto exit;

    err = DNSServiceBrowse( &m_serviceRef, 0, 0, "_http_tcp", NULL, BrowseReply, this );

    if ( err )
        goto exit;

    err = WSAAsyncSelect( (SOCKET) DNSServiceRefSockFD(m_serviceRef), m_hWnd,
                        WM_PRIVATE_SERVICE_EVENT, FD_READ|FD_CLOSE);

exit:

    if ( err )
    {
        if ( m_serviceRef )
        {
            DNSServiceRefDeallocate( m_serviceRef );
            m_serviceRef = NULL;
        }
    }

    return( err );
}

void MyWindow::OnDestroy( void )
{
    // ...

    if ( m_serviceRef )
    {
        WSAAsyncSelect( (SOCKET) DNSServiceRefSockFD( m_serviceRef ), m_hWnd, 0, 0 );
        DNSServiceRefDeallocate( m_serviceRef );
    }

    // ...
}
```

Example 7-10. Windows MFC example (continued)

```
LONG MyWindow::OnServiceEvent(WPARAM inWParam, LPARAM inLParam)
{
    SOCKET          sock = (SOCKET) inWParam;
    DNSServiceErrorType err;

    if ( WSAGETSELECTERROR(inLParam) && !(HIWORD(inLParam)))
        goto exit;

    ASSERT( ( SOCKET ) DNSServiceRefSockFD( m_serviceRef ) == sock );
    err = DNSServiceProcessResult( m_serviceRef );
    ASSERT( !err );

exit:

    return ( 0 );
}

void DNSSD_API MyWindow::BrowseReply(
    DNSServiceRef          inRef,
    DNSServiceFlags        inFlags,
    uint32_t               inInterfaceIndex,
    DNSServiceErrorType    inErrorCode,
    const char *           inName,
    const char *           inType,
    const char *           inDomain,
    void *                 inContext )
{
    MyWindow * self = reinterpret_cast<MyWindow*>( inContext );
    ASSERT( self );

    // ...
}
```

Event Handling with Independent Threads

DNS-SD also accommodates programmers who prefer to use multiple threads rather than a single thread and an event loop. Because the `DNSServiceProcessResult()` blocks if no data is available, you can simply create a thread and have it spin calling `DNSServiceProcessResult()`. When no data is available, the thread will sleep. When data arrives, the thread will wake up, handle it, and then go back to sleep again. Example 7-11 shows how to set up event handling using independent threads.

Example 7-11. Event handling with independent threads

```
void ThreadProc(DNSServiceRef ref)
{
    while (DNSServiceProcessResult(ref) == kDNSServiceErr_NoError)
        continue;
}
```

This has the advantage that your callback routines will get executed “by magic,” without you having to take any special action aside from creating the thread and starting it running in the first place. Of course, magic comes at a price. When callback routines get executed by magic, you no longer have control over exactly when they will run and what else might be happening at the same time. As with all multi-threaded code, you need to take good care to use the proper locking to avoid race conditions and crashes.

Summary

This chapter introduced the lowest level, cross-platform, C-programming API for Zeroconf’s DNS Service Discovery. It also introduced the important DNS-SD concepts of asynchronous event handling and continuous live updating, which apply no matter which language or API you choose to use. The following chapters present other APIs and languages. In some, callbacks are scheduled preemptively using threads, and you need to take care to write thread-safe code. In others, callbacks are scheduled cooperatively out of a main event loop, and you don’t have to worry about thread safety, reentry, and race conditions, but in your callback functions and methods you need to be careful to avoid doing time-consuming operations. If your callback function or method does anything that blocks for a long period of time, you’ll cause your application’s whole user interface to lock up while it waits for your callback to finish what it’s doing.