NSNetServices and CFNetServices Programming Guide

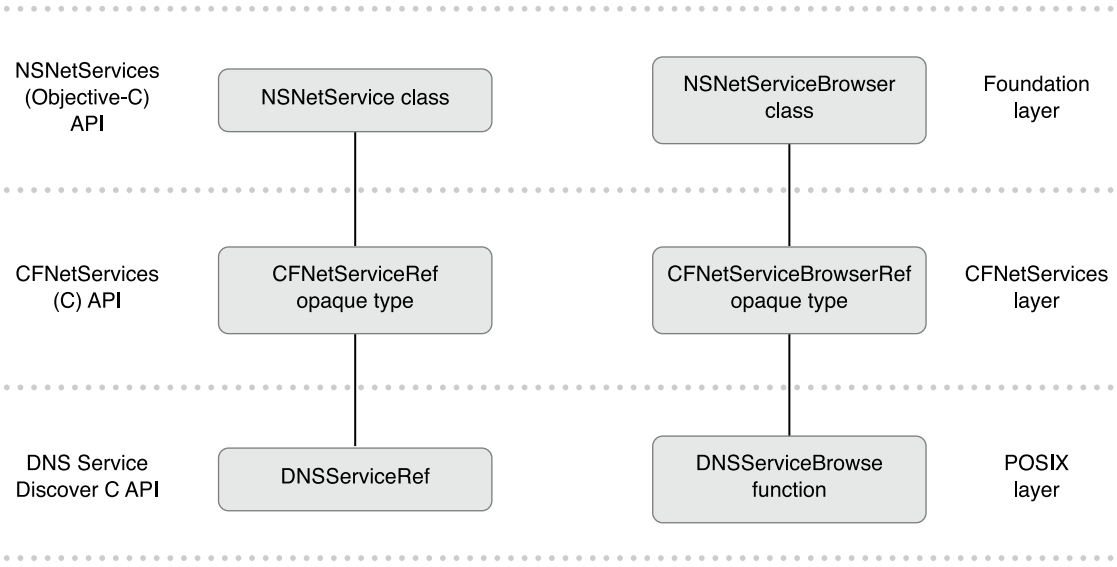
Developer

Contents

Figures, Tables, and Listings

About NSNetServices and CFNetServices

The `NSNetService` class and the `CFNetServices` C API provide high-level abstractions for advertising, browsing, discovering, and resolving Bonjour services. After publishing or discovering a service, your app is responsible for providing networking code to do the actual communication.



At a Glance

Both the `NSNetService` class and the `CFNetServices` C API are based on run loops, and can be integrated with your own networking code written using CFNetwork or Foundation networking APIs.

The `NSNetService` class is a Cocoa class that provides easy integration with GUI apps, Foundation run loops, and the `NSStream` family of networking classes. If you are writing Bonjour code to interface with code at this level, you should generally use the `NSNetService` class.

The `CFNetServices` API (described in *CFNetServices Reference*) is a CFNetwork-based class that provides easy integration with the CFNetwork family of networking APIs. If you are writing Bonjour code to interface with Core Foundation–level code, you can use either the `CFNetServices` API or the `NSNetService` class.

Note: In addition to the APIs described above, Bonjour provides a POSIX API that is outside the scope of this document. To learn more about this low-level API for adding Bonjour functionality to BSD-style software, read *DNS Service Discovery Programming Guide*.

How to Use This Document

Whether you want to use the `NSNetService` class or the `CFNetServices` API, read "Foundation Network Services Architecture" (page 8) to gain an understanding of the Foundation classes available. These Foundation classes map fairly neatly onto `CFNetServices` opaque types; only the names and calling conventions differ.

Next, decide what Bonjour tasks your app needs to perform and read the appropriate chapter or chapters for the Foundation-level API (even if you intend to use the `CFNetServices` API). These chapters provide conceptual information about how to perform each task, along with code snippets based on the Foundation-level API.

Next, if you want to use the lower-level CFNetwork-based API, read "Using CFNetServices" (page 37), which provides code snippets that show you how to do so.

Finally, if you want to provide additional user control over which domains your app uses when browsing for or advertising services, read the appendix, "Browsing for Domains" (page 47).

Prerequisites

This document assumes that you are familiar with Bonjour, and have already read *Bonjour Overview*. This document also assumes that you are familiar with OS X and iOS networking as a whole, including the concepts described in *Networking Overview*.

Foundation Network Services Architecture

The architecture for Bonjour network services in the Foundation framework is designed so that you do not need to understand the details of DNS record management, instead letting you manage services in terms of the four fundamental operations defined for Bonjour network services:

- Advertising a service, a process known as *publication*
- Browsing for available services, also called *service discovery*
- Connecting to services directly (the preferred technique)
- Resolving services (translating service names into something that lets you connect to them)

Foundation Service Discovery Classes

To support these operations, the Foundation framework defines two classes—one that you use to browse for services and domains, and one that represents an individual service.

NSNetServiceBrowser

The `NSNetServiceBrowser` class has two purposes: browsing for services and browsing for Bonjour domains. At any given time, a single `NSNetServiceBrowser` object can execute at most one search operation; if you need to search for multiple types of services at once, use multiple `NSNetServiceBrowser` objects.

When browsing for a specific type of service—for example, searching for FTP services on the local network—this class provides you with filled-out instances of `NSNetService` for each remote service.

When browsing for domains, whether for registration or browsing, this class provides you with a list of available domain names. (Note that domain browsing is outside the scope of this chapter. For step-by-step instructions and code examples about domain browsing, and guidance about how to use browsing in your code, read "Browsing for Domains" (page 47).)

NSNetService

The `NSNetService` class represents a single service (either a remote service that your app wants to use or a local service that your app is publishing). Instances of this class are involved in all of the Bonjour operations described in this chapter.

`NSNetService` objects that represent remote services come preinitialized with the service name, type, and domain, but no host name, IP address or port number. The name, type, and domain are used to resolve the instance into the information needed to connect to the service (IP addresses and port number) upon request.

`NSNetService` objects used for publishing local services to the network must be initialized (by your code) with the service name, type, domain, and port number. Bonjour uses this information to announce the availability of your service over the network.

Theory of Operation

Because network discovery can take a long time to complete, the `NSNetService` and `NSNetServiceBrowser` classes perform all their operations asynchronously. The methods provided by `NSNetService` and `NSNetServiceBrowser` return immediately. As a result, your app can continue executing while network operations take place.

Both classes require delegate objects in your app that must implement appropriate methods to handle the resulting data. In some cases, these classes may call your delegate methods more than once, providing additional data with each call.

Publication

The `NSNetService` class handles service publication. To publish a service:

1. The app sets up a socket and begins listening for incoming connections on that socket, as described in *Networking Programming Topics*.
2. The app initializes an `NSNetService` object, providing the port number, the name of the service, and domain information, and then sets up a delegate object to receive results. The object automatically schedules itself with the current run loop.
3. The app sends a message to the `NSNetService` object, requesting that the service be published to the network, and publication proceeds asynchronously.
4. The `NSNetService` object calls its delegate with information about the status of the publication.

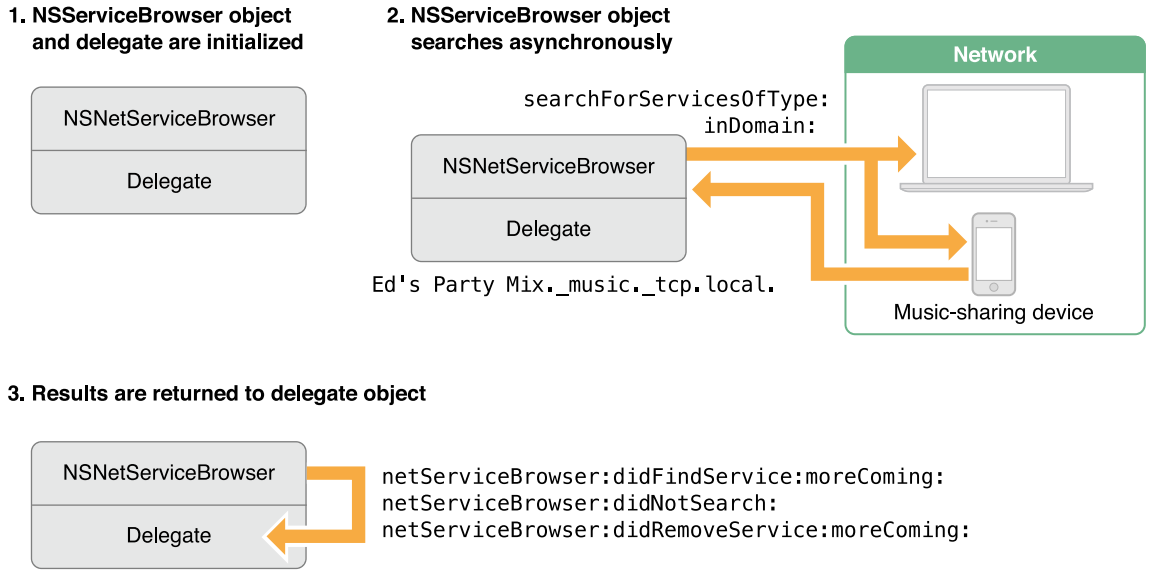
When the service has been successfully published, the delegate is notified. If publication fails for any reason, the delegate is also notified, along with appropriate error information. If publication proceeds successfully, no further messages are sent to the delegate.

For step-by-step instructions and code examples about service publication, see "Publishing Network Services" (page 14).

Service Discovery

To discover services advertised on the network, you use the `NSNetServiceBrowser` class, as shown in Figure 1-1.

Figure 1-1 Service discovery with `NSNetServiceBrowser`



In step 1, the app initializes an `NSNetServiceBrowser` object and associates a delegate object with it. In step 2, the `NSNetServiceBrowser` object searches asynchronously for services. In step 3, results are returned to the delegate object in the form of `NSNetService` objects.

Step 3 usually occurs multiple times. Initially, the browser calls you once for each service that it currently knows about. When it calls your delegate method with the last of these currently known services, it passes `NO` for the `moreComing` parameter.

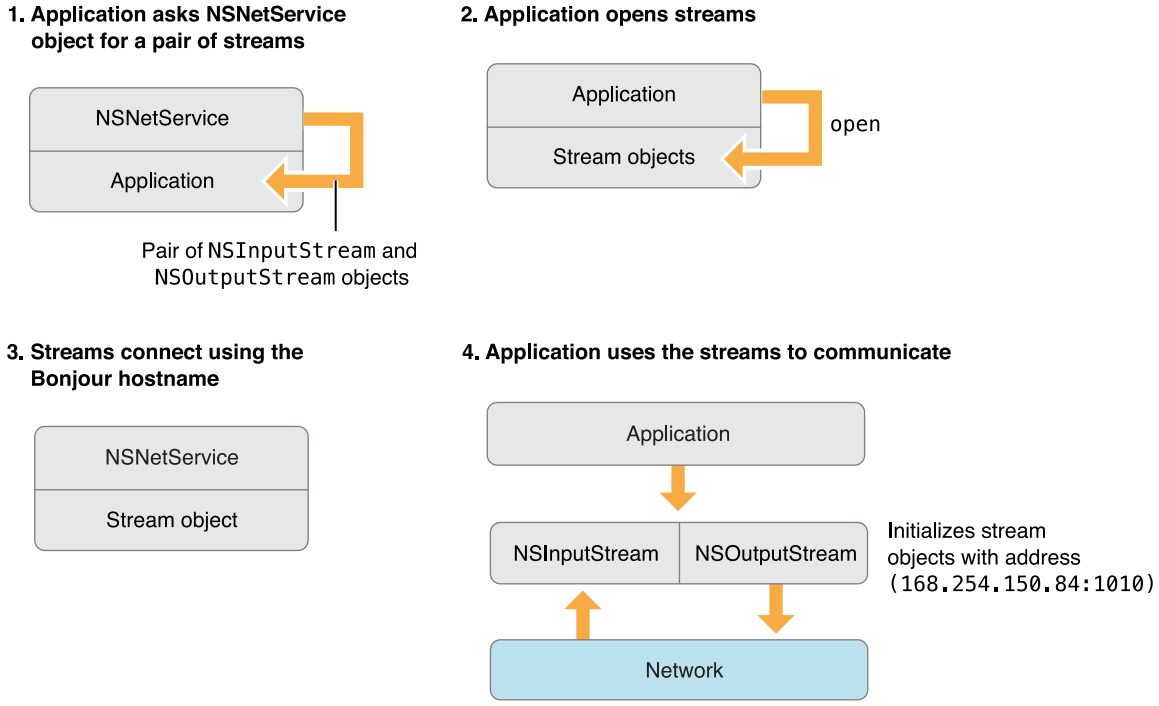
However, because the browser object continues to browse until your app explicitly tells it to stop, your delegate learns about new services as they come online and learns about the disappearance of services as they shut down. As a result, your `didFindService:moreComing:` method may be called with new services even after you get a `moreComing` value of `NO`.

For step-by-step service discovery instructions and sample code snippets, read "Browsing for Network Services" (page 19).

Connecting to Services

To connect to a Bonjour-advertised service, an app typically calls a method such as `getInputStream:outputStream:`, which provides a stream or pair of streams for communicating with the service (as shown in Figure 1-2). When the app opens that stream, the stream object itself connects to the Bonjour service in the same way that it would connect if you had passed it a hostname. The stream object, in turn, uses the `NSNetService` object to perform DNS lookups on its behalf.

Figure 1-2 Connecting to a Bonjour Service (`NSNetService`)



Although the steps above describe the recommended way to connect to a Bonjour-advertised service, you can also connect in two other ways:

- By resolving the `NSNetService` object, and then asking the `NSNetService` object to provide the service's hostname, then passing that hostname to a connect-by-name API.
- By asking the `NSNetService` object to resolve the service to a set of IP addresses (as described in "Resolving Services Manually" (page 12)), and then attempting to connect to the host yourself. This technique is discouraged, however, because of the complexity caused by multihoming.

For step-by-step service connection instructions and sample code snippets, read "Connecting to and Monitoring Network Services" (page 25).

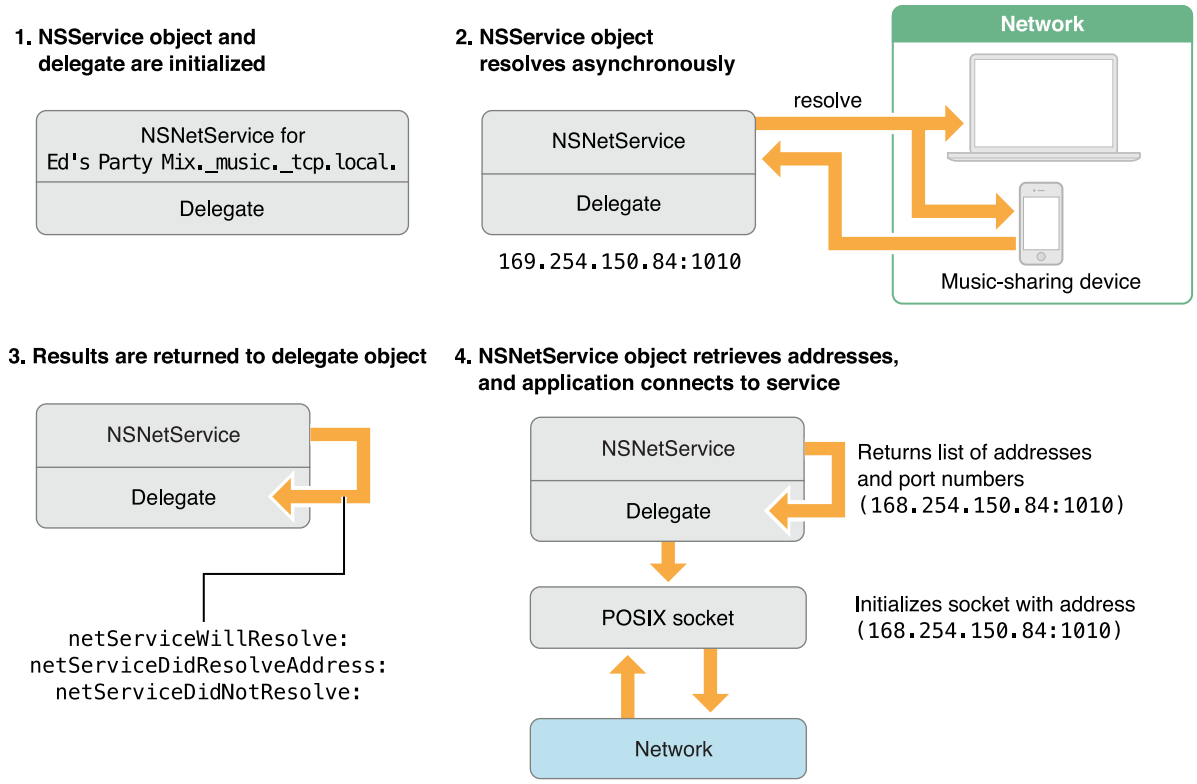
Resolving Services Manually

In the most common usage, resolution is performed transparently, either when you connect to a service by calling a connect-to-service method such as `getInputStream:outputStream:` or when you later reconnect to a service using a name obtained by calling `hostName` on an already resolved service object.

If for some reason you need to know the actual IP addresses for a service, you can also ask the object to resolve a service name explicitly. To prevent your app from slowing down, resolution takes place asynchronously, returning results or error messages to the `NSNetService` object's delegate. If valid addresses were found for the service, your app can use them to make a socket connection.

Figure 1-3 illustrates this process.

Figure 1-3 Service resolution with `NSNetService`



In step 1, the app either explicitly initializes or otherwise obtains an `NSNetService` instance for the service—in this case a local `music` service over TCP called `Ed's Party Mix`. In step 2, the `NSNetService` object receives a `resolve` message. The resolution proceeds asynchronously, and at some point it receives an IP address and port number for the service (`169.254.150.84:1010`). In step 3, the delegate is notified, and in step 4, the delegate asks the `NSNetService` object for a list of addresses and a port number. A service can have multiple IP addresses, but always has exactly one port.

For step-by-step service resolution instructions and sample code snippets, read "Connecting to and Monitoring Network Services" (page 25).

Publishing Network Services

Bonjour enables dynamic discovery of network services on IP networks without a centralized directory server. The Foundation framework's `NSNetService` class represents instances of Bonjour network services. This chapter describes the process for publishing Bonjour network services with `NSNetService`.

The Publication Process

Bonjour network services use standard DNS information to advertise their existence to potential clients on a network. In Cocoa, the `NSNetService` class handles the details of service publication.

Typically, you use `NSNetService` to publish a service provided by a socket owned by the same process. However, because the `NSNetService` class does not use the socket in any way, you can also use the class to advertise on behalf of another process's service, such as an FTP server process that has not yet been updated to support Bonjour. However, if you are creating an IP network service, you should include Bonjour publication code as part of its startup process.

Because network activity can sometimes take some time, `NSNetService` objects process publication requests asynchronously, delivering information through delegate methods. To use `NSNetService` correctly, your app must assign a delegate to each `NSNetService` instance it creates. Because the identity of the `NSNetService` object is passed as a parameter in delegate methods, you can use one delegate for multiple `NSNetService` objects.

Publishing a Bonjour network service takes four steps:

1. Set up a valid TCP listening socket (or a UDP data socket) for communication.
2. Initialize an `NSNetService` instance with name, type, domain, and port number, and assign a delegate to the object.
3. Publish the `NSNetService` instance.
4. Respond to messages sent to the `NSNetService` object's delegate.

The following sections describe these steps in detail.

> **Note:** Although Bonjour requires you to create a socket for your app's communication, Bonjour does not use this socket. It is solely used by the networking code that you provide. If you are adding Bonjour support to an existing network app and already have a socket configured to listen for incoming requests, you can ignore step 1.

Configuring a Socket for Your Service

Bonjour network services require either a TCP listening socket or a UDP data socket.

> **Important:** If you are publishing a service with both IPv4 and IPv6 addresses, the port number *must* be the same for IPv4 and IPv6.

If you already have existing networking code, you can continue using it as is, and provide its port number to Bonjour when you initialize the service.

If you don't have existing networking code, use the CFSocket API:

- For TCP, use a `CFSocketRef` object to listen for incoming connections, then use `NSStream` or `CFStreamRef` object for the actual communication. For good examples of how to set up a network listener, see the *RemoteCurrency*, *CocoaEcho*, and *SimpleNetworkStreams* sample code projects.
- For UDP, use a `CFSocketRef` object for sending and receiving packets. For an example, see the *UDPEcho* sample code project.

To learn why CFSocket is recommended, read "Using Sockets and Socket Streams" in *Networking Overview*.

For a more detailed overview of how to write a TCP- or UDP-based daemon, read "Using Sockets and Socket Streams" in *Networking Programming Topics*.

Initializing and Publishing a Network Service

To initialize an `NSNetService` instance for publication, use the `initWithDomain:type:name:port:` method. This method sets up the instance with appropriate socket information and adds it to the current run loop.

The service type expresses both the application-layer protocol (FTP, HTTP, and so on) and the transport protocol (TCP or UDP). The format is as described in "Domain Naming Conventions", for example, `_printer._tcp` for a printer over TCP.

The service name can be an arbitrary `NSString`, but the value must be no longer than 63 bytes in UTF-8 encoding. Because this is the name that should be presented to users, it should be human-readable and descriptive of the specific service instance. Consider letting the user override any default name that you provide.

One recommended approach is to use the computer name as the service name. If you pass the empty string (`@""`) for the service name parameter, the system automatically advertises your service using the computer name as the service name. For examples of other naming approaches, read *Bonjour Overview*.

If you need to construct the service name in a nonstandard way, you can retrieve the computer name yourself. In OS X on the desktop, you can obtain the computer name by calling the `SCDynamicStoreCopyComputerName` function from the System Configuration framework. In iOS, you can obtain the same information from the `name` property of the `UIDevice` class.

> **Note:** By default, the specified service name is only a hint. If that service name conflicts with an existing service on the network, Bonjour chooses a new name.
>
> If you need to guarantee an exact service name, you must use the `publishWithOptions:` method to publish your service and set the `NSNetServiceNoAutoRename` flag in the options parameter. With this flag set, if the specified name conflicts with an existing service, your delegate's `netService:didNotPublish:` method is called with the error code `NSNetServicesCollisionError`.

When publishing a service, you must also specify the domain in which the service should be published. Here are some common values:

- `@""`—Registers the service in the default set of domains. Pass this value unless you have a specific reason not to.
- `@"local"`—Registers the service *only* on the local network. Pass this value if you need to prevent publishing your service over Back to My Mac or wide-area Bonjour.
- A user-specified domain—Registers the service in only the specified domain. To retrieve a list of existing domains, call the `searchForRegistrationDomains` method (as described in "Browsing for Domains" (page 47)) or allow the user to enter an arbitrary domain name.

Upon initialization, the `NSNetService` object is automatically scheduled on the current run loop with the default mode. If you want to schedule it on a different run loop or with a different mode, you can call the `removeFromRunLoop:forMode:` and `scheduleInRunLoop:forMode:` methods.

After the initialization is complete and valid, assign a delegate to the `NSNetService` object with the `setDelegate:` method. Finally, publish the service with the `publish` method, which returns immediately. Bonjour performs the publication asynchronously and returns results through delegate methods.

Listing 2-1 demonstrates the initialization and publication process for Bonjour network services. An explanation of the code follows it. For a good example of service publication, see the *PictureSharing* sample code project in the Mac Developer Library.

Listing 2-1 Initializing and publishing a Bonjour network service

```
void myRegistrationFunction(uint16_t port) {
id delegateObject;      // Assume this exists.
NSNetService *service;

    service = [[NSNetService alloc] initWithDomain:@""                          // 1
                                    type:@"_music._tcp"
                                    name:@""
                                    port:port];
    if(service)
    {
        [service setDelegate:delegateObject];                                   // 2
        [service publish];                                                      // 3
    }
    else
    {
        NSLog(@"An error occurred initializing the NSNetService object.");
    }
}
```

Here's what the code does:

1. Initializes the `NSNetService` object. This example uses the default domain(s) for publication and a hypothetical TCP/IP music service.
2. Sets the delegate for the `NSNetService` object. This object handles all results from the `NSNetService` object, as described in "Implementing Delegate Methods for Publication" (page 18).
3. Publishes the service to the network.

To stop a service that is already running or is in the process of starting up, use the `stop` method.

Implementing Delegate Methods for Publication

`NSNetService` provides your app with publication status information by calling methods on its delegate. If you are publishing a service, your delegate object should implement the following methods:

```
netServiceWillPublish:
netServiceDidPublish:
netService:didNotPublish:
netServiceDidStop:
```

The `netServiceWillPublish:` method notifies the delegate that Bonjour is ready to publish the service. When this method is called, the service is not yet visible to the network, which means that publication may still fail. However, you can assume that the service is visible unless `NSNetService` calls your delegate's `netService:didNotPublish:` method.

The `netServiceDidPublish:` method notifies the delegate that Bonjour has successfully published the service.

> **Important:** As mentioned earlier in this chapter, unless you specifically disable service renaming, your service is automatically renamed if it conflicts with the name of an existing service on the network. If your app depends on the name of the service (whether because its clients store service names for later use or because you display the service name in the user interface), your `netServiceDidPublish:` method should call the service object's `name` method to obtain the name that was actually published.
>
> If your app's potential clients store service names to allow them to reconnect to your app at a later time, your app should save the modified name into its preferences file. When the user runs your app in the future, your app should create an `NSNetService` object with the modified name so clients can reconnect. For more details, read "Initializing and Publishing a Network Service" (page 15).

The `netService:didNotPublish:` method is called when publication fails for any reason. Your delegate's `netService:didNotPublish:` method should extract the type of error from the returned dictionary using the `NSNetServicesErrorCode` key and handle the error accordingly. For a complete list of possible errors, see *NSNetService Class Reference*.

The `netServiceDidStop:` method gets called as a result of the `stop` message being sent to the `NSNetService` object. If this method gets called, the service is no longer published.

Browsing for Network Services

An important part of Bonjour is the ability to browse for services on the network. This chapter describes how to use the `NSNetServiceBrowser` class to discover Bonjour network services.

The Browsing Process

The `NSNetServiceBrowser` class provides methods for browsing for available Bonjour network services.

Because of the possible delays associated with network traffic, `NSNetServiceBrowser` objects perform browsing asynchronously by registering with the default run loop. Browsing results are returned to your app through delegate methods. To handle results from an `NSNetServiceBrowser` object, you must assign it a delegate.

To Browse for Bonjour network services, your app must perform three steps:

1. Initialize an `NSNetServiceBrowser` instance and assign a delegate to the object.
2. Begin a search for services of a specific type in a given domain.
3. Handle search results and other messages sent to the delegate object.

Initializing the Browser and Starting a Search

To initialize an `NSNetServiceBrowser` object, use the `init` method. This sets up the browser and adds it to the current run loop. If you want to use a run loop other than the current one, use the `removeFromRunLoop:forMode:` and `scheduleInRunLoop:forMode:` methods.

To begin the browsing process, use the `searchForServicesOfType:inDomain:` method. The service type expresses both the application-layer protocol (FTP, HTTP, and so on.) and the transport protocol (TCP or UDP). The format is as described in "Domain Naming Conventions"—for example, `_printer._tcp` for an LPR printer over TCP.

The domain parameter specifies the DNS domain in which the browser performs its search. There are four common ways to use this parameter:

- To limit searches to the local network (for a chat program, for example), pass `@"local"` to search only the local LAN.
- For limited wide-area support, pass `@""` to search the system's default search domains. To avoid confusion, be sure to display each service's domain in your user interface.
- For full wide-area support, use the `searchForBrowsableDomains` method to obtain a list of browsable domains, as described in "Browsing for Domains" (page 47). Then present a user interface that allows the user to choose the domain to browse. When the user chooses a domain, browse only in that domain.
- To provide maximum flexibility for power users, provide a text field in which the user can specify an arbitrary domain name to browse.

To stop a search, use the `stop` method. Then perform any necessary cleanup in the `netServiceBrowserDidStopSearch:` delegate callback.

Listing 3-1 demonstrates how to browse for Bonjour network services with `NSNetServiceBrowser`. The code initializes the object, assigns a delegate, and begins a search for a hypothetical music service type, `_music._tcp`, on the local network. For a good example of service browsing, see the *PictureSharingBrowser* sample code project in the Mac Developer Library.

Listing 3-1 Browsing for Bonjour network services

```
id delegateObject; // Assume this exists.
NSNetServiceBrowser *serviceBrowser;

serviceBrowser = [[NSNetServiceBrowser alloc] init];
[serviceBrowser setDelegate:delegateObject];
[serviceBrowser searchForServicesOfType:@"_music._tcp" inDomain:@""];
```

Implementing Delegate Methods for Browsing

`NSNetServiceBrowser` returns all browsing results to its delegate. If you are using the class to browse for services, your delegate object should implement the following methods:

`netServiceBrowserWillSearch:`

`netServiceBrowserDidStopSearch:`

`netServiceBrowser:didNotSearch:`

`netServiceBrowser:didFindService:moreComing:`

```
netServiceBrowser:didRemoveService:moreComing:
```

The `netServiceBrowserWillSearch:` method notifies the delegate that a search is commencing. You can use this method to update your user interface to reflect that a search is in progress. When browsing stops, the delegate receives a `netServiceBrowserDidStopSearch:` message. In that delegate method, you can perform any necessary cleanup.

If the delegate receives a `netServiceBrowser:didNotSearch:` message, the search failed for some reason. The delegate method should extract the error information from the dictionary with the `NSNetServicesErrorCode` key and handle the error accordingly. For a list of possible errors, see NSNetService.

You track services with the `netServiceBrowser:didFindService:moreComing:` and `netServiceBrowser:didRemoveService:moreComing:` methods, which indicate that a service has become available or has shut down. The `moreComing` parameter indicates whether more results are on the way. If this parameter is `YES`, delay updating any user interface elements until the method is called with a `moreComing` parameter of `NO`.

Important: A `moreComing` value of `NO` does *not* indicate that browsing has finished. It indicates that your app has been provided with the complete state of the world as Bonjour understands it up to that point in time. There is still a chance that more services will become available as new services appear on the network or as unusually slow devices respond to requests.

Be sure to retain the `didFindService` parameter before trying to resolve it. Otherwise, you risk the object becoming deallocated. If you want a list of available services, you must maintain your own array based on the information provided by delegate methods.

Note: If you specify the `@""` search domain, you may receive search results with the same name in multiple domains. For example, you might get `"Foo._music._tcp.local."` and `"Foo._music._tcp.arbitraryusername.me.com."`. To avoid causing confusion, consider designing your user interface so that it restricts the results to the local domain, groups the results by domain, or shows the domain information in a separate column.

Listing 3-2 shows the interface for a class that responds to the `NSNetServiceBrowser` delegate methods required for service browsing, and Listing 3-3 shows its implementation. You may want to use this code as a starting point for your service browsing code.

Listing 3-2 Interface for an NSNetServiceBrowser delegate object used when browsing for services

```
#import <Foundation/Foundation.h>
```

```
@interface NetServiceBrowserDelegate : NSObject <NSNetServiceBrowserDelegate>
{
}

@property BOOL searching;
@property (strong,atomic) NSMutableArray *services;

// Other methods
- (void)handleError:(NSNumber *)error;
- (void)updateUI;

@end
```

Listing 3-3 Implementation for an NSNetServiceBrowser delegate object used when browsing for services

```
#import "NetServiceBrowserDelegate.h"

@implementation NetServiceBrowserDelegate

- (id)init
{
    self = [super init];
    if (self) {
        self.services = [NSMutableArray arrayWithCapacity: 0];
        self.searching = NO;
    }
    return self;
}

// Sent when browsing begins
- (void)netServiceBrowserWillSearch:(NSNetServiceBrowser *)browser
{
    self.searching = YES;
    [self updateUI];
}
```

```
// Sent when browsing stops
- (void)netServiceBrowserDidStopSearch:(NSNetServiceBrowser *)browser
{
    self.searching = NO;
    [self updateUI];
}

// Sent if browsing fails
- (void)netServiceBrowser:(NSNetServiceBrowser *)browser
        didNotSearch:(NSDictionary *)errorDict
{
    self.searching = NO;
    [self handleError:[errorDict objectForKey:NSNetServicesErrorCode]];
}

// Sent when a service appears
- (void)netServiceBrowser:(NSNetServiceBrowser *)browser
        didFindService:(NSNetService *)aNetService
        moreComing:(BOOL)moreComing
{
    [self.services addObject:aNetService];
    NSLog(@"Got service %p with hostname %@\n", aNetService,
        [aNetService hostName]);

    if(!moreComing)
    {
        [self updateUI];
    }
}

// Sent when a service disappears
- (void)netServiceBrowser:(NSNetServiceBrowser *)browser
        didRemoveService:(NSNetService *)aNetService
        moreComing:(BOOL)moreComing
```

```
{
    [self.services removeObject:aNetService];

    if(!moreComing)
    {
        [self updateUI];
    }
}

// Error handling code
- (void)handleError:(NSNumber *)error
{
    NSLog(@"An error occurred. Error code = %d", [error intValue]);
    // Handle error here
}

// UI update code
- (void)updateUI
{
    if(self.searching)
    {
        // Update the user interface to indicate searching
        // Also update any UI that lists available services
    }
    else
    {
        // Update the user interface to indicate not searching
    }
}

@end
```

Connecting to and Monitoring Network Services

This chapter describes how to connect to a Bonjour-advertised service. There are two occasions when you might need to connect to a Bonjour service:

- You have just browsed for services, and the user has chosen one.
- You have saved information about a service for later use, and you have explicitly initialized an instance of the `NSNetService` class with that information.

In either case, the way you connect to the service is the same. There are three ways to connect to a service, listed in order of preference:

- Using a stream-based connect-to-service method such as `getInputStream:outputStream:`.
- Connecting by hostname.
- Connecting by IP address.

These techniques are described in the sections that follow. In addition, the section "Persisting a Bonjour Service" (page 34) describes how to properly store information about a service for later reuse, and the section "Monitoring a Bonjour Service" (page 35) explains how a Bonjour service can provide additional information that clients can passively monitor without connecting to the service.

Connecting with Streams

If you are connecting using Foundation's raw TCP stream API (as opposed to a higher-level API, such as `NSURLConnection`), the most straightforward way to connect to a service is with a connect-to-service API, such as `getInputStream:outputStream:`.

This method provides a reference to an input stream (`NSInputStream`) and an output stream (`NSOutputStream`), both of which you may access synchronously or asynchronously. To interact asynchronously, you must schedule the streams in the current run loop and assign them a delegate object. These streams provide a general-purpose means of communicating with TCP-based network services that is not tied to any specific protocol (such as HTTP).

Listing 4-1 demonstrates how to connect to a network service (`NSNetService`) using streams. Note that if you require only one of the two streams, you should pass `NULL` for the other parameter so that you do not get the other stream back. For a complete example of service resolution using `NSStream` objects, see the *PictureSharing* sample code project in the Mac Developer Library.

Listing 4-1 Connecting to a resolved Bonjour network service

```
#import <sys/socket.h>
#import <netinet/in.h>

// ...

NSNetService *service;  // Assume this exists. For instance, you may
                        // have received it from an NSNetServiceBrowser
                        // delegate callback.

NSInputStream *istream = nil;
NSOutputStream *ostream = nil;

[service getInputStream:&istream outputStream:&ostream];
if (istream && ostream)
{
    // Use the streams as you like for reading and writing.
}
else
{
    NSLog(@"Failed to acquire valid streams");
}
```

Connecting by Name

If you are connecting to a Bonjour service using a higher-level API such as `NSURLConnection`, you should connect using the service's hostname.

You can obtain the hostname for a Bonjour service by resolving the service and then calling the `hostName` method on the `NSNetService` object. Then, pass that hostname and port to the appropriate API, just as you would any other hostname.

The Resolution Process

When Bonjour resolves a service, it does two things:

- Looks up the service name information to get a hostname and port number.
- Looks up the hostname to provide a set of IP addresses.

Because resolution can take time, especially if the service is unavailable, `NSNetService` resolves asynchronously, providing information to your app through a delegate object.

To resolve and use an `NSNetService` instance, your app must do four things:

1. Obtain an `NSNetService` instance through initialization or service discovery.
2. Resolve the service.
3. Respond to messages sent to the object's delegate about addresses or errors.
4. Use the resulting addresses or hostname to connect to the service.

Obtaining and Resolving an NSNetService Object

You can obtain an `NSNetService` object representing the service you want to connect to in one of two ways:

- Use `NSNetServiceBrowser` to discover services.
- Initialize a new `NSNetService` object with the name, type, and domain of a service that is known to exist, usually saved from a previous browsing session.

For information about service browsing, see "Browsing for Network Services" (page 19).

To create an `NSNetService` object for resolution rather than publication, use the `initWithDomain:type:name:` method.

> **Note:** Although you can use an `NSNetService` object to browse without specifying a domain, a service object *must* have a specific domain name when resolving a service. This domain is usually obtained from a previous browsing session (or stored from a previous browsing session).

Once you have an `NSNetService` object to resolve, assign it a delegate and use the `resolveWithTimeout:` method to asynchronously resolve the service. When resolution is complete, the delegate receives a `netServiceDidResolveAddress:` message upon success or a `netService:didNotResolve:` message if an error occurred. Because the delegate receives the identity of the `NSNetService` object as part of the delegate method, one delegate can serve multiple `NSNetService` objects.

Listing 4-2 demonstrates how to initialize and resolve an `NSNetService` object for a hypothetical music-sharing service. The code initializes the object with the name `serviceName`, with the type `_music._tcp`, and with the link-local suffix `local.`. It then assigns it a delegate and asks it to resolve the name into socket addresses.

Listing 4-2 Resolving network services with NSNetService

```
id delegateObject = [[NetServiceResolutionDelegate alloc] init]; // Defined below
NSString *serviceName = ...;
NSNetService *service;

service = [[NSNetService alloc] initWithDomain:@"local." type:@"_music._tcp"
                                name:serviceName];
[service setDelegate:delegateObject];
[service resolveWithTimeout:5.0];
```

Implementing Delegate Methods for Resolution

`NSNetService` returns resolution results to its delegate. If you are resolving a service, your delegate object should implement the following methods:

`netServiceDidResolveAddress:`

`netService:didNotResolve:`

Assuming that you are connecting by hostname, you can request that information as soon as your delegate's `netServiceDidResolveAddress:` is first called. Be careful, though, because this method can be called more than once. (The reason is explained further in "Connecting to a Bonjour Service by IP Address" (page 34).)

If resolution fails for any reason, the `netService:didNotResolve:` method is called. If the delegate receives a `netService:didNotResolve:` message, you should extract the type of error from the returned dictionary using the `NSNetServicesErrorCode` key and handle the error accordingly. For a list of possible errors, see *NSNetService Class Reference*.

Listing 4-3 shows the interface for a class that acts as a delegate for multiple `NSNetService` objects, and Listing 4-4 shows its implementation. You can use this code as a starting point for more sophisticated tracking of resolved services.

Listing 4-3 Interface for an `NSNetService` delegate object used when resolving the service

```
#import <Foundation/Foundation.h>

@interface NetServiceResolutionDelegate : NSObject <NSNetServiceDelegate>
{
    // Keeps track of services handled by this delegate
    NSMutableArray *services;
}

// Other methods
- (BOOL)addressesComplete:(NSArray *)addresses
        forServiceType:(NSString *)serviceType;
- (void)handleError:(NSNumber *)error withService:(NSNetService *)service;

@end
```

Listing 4-4 Implementation for an `NSNetService` delegate object (resolution)

```
#import "NetServiceResolutionDelegate.h"

@implementation NetServiceResolutionDelegate

- (id)init
{
    self = [super init];
    if (self) {
        services = [[NSMutableArray alloc] init];
    }
```

```
    return self;
}

// Sent when addresses are resolved
- (void)netServiceDidResolveAddress:(NSNetService *)netService
{
    // Make sure [netService addresses] contains the
    // necessary connection information
    if ([self addressesComplete:[netService addresses]
            forServiceType:[netService type]]) {
        [services addObject:netService];
    }
}

// Sent if resolution fails
- (void)netService:(NSNetService *)netService
        didNotResolve:(NSDictionary *)errorDict
{
    [self handleError:[errorDict objectForKey:NSNetServicesErrorCode]
withService:netService];
    [services removeObject:netService];
}

// Verifies [netService addresses]
- (BOOL)addressesComplete:(NSArray *)addresses
        forServiceType:(NSString *)serviceType
{
    // Perform appropriate logic to ensure that [netService addresses]
    // contains the appropriate information to connect to the service
    return YES;
}

// Error handling code
- (void)handleError:(NSNumber *)error withService:(NSNetService *)service
{
```

```
    NSLog(@"An error occurred with service %@.%@.%@, error code = %d",
        [service name], [service type], [service domain], [error intValue]);
    // Handle error here
}

@end
```

When Resolving Fails

In networking, it is not possible to have perfect knowledge about the state of the world. All you can know with certainty is what packets you have already received. As a result, your app might think that a service is available when it isn't, or it might think a service is not available when it is.

Because failing to show a valid service is a bigger problem for the user than showing a stale service, Bonjour deliberately errs on the side of assuming that a service is still available. This means that there are many ways a service can go away without your app immediately being aware of it:

- A device can be abruptly switched off, or the power can fail.
- A device can be unplugged from a wired network.
- A device can move out of a wireless base station's range.
- Packet loss can prevent the "goodbye" packet from reaching its destination.

As a result, although Bonjour generally discovers new services within a few seconds, if a service goes away, the disappearance of the service may not be discovered until your app tries to connect to it and gets no response.

You should not assume that just because the Bonjour APIs report a discovered service, the service is guaranteed to be available when the software tries to access it. All discovery means is that at some time in the recent past, Bonjour received packets indicating that the service did exist, not that the service necessarily exists right now.

To minimize the impact of stale records:

- Don't cancel resolving prematurely. Allow resolution to continue until the app successfully connects or the user cancels the attempt. An actively running service resolution serves as a hint to Bonjour that the named service does not seem to be responding.
- If service resolution returns a result but the client is unable to open a TCP connection to it within 5–10 seconds, low-level C clients should call `DNSServiceReconfirmRecord`. This call tells Bonjour that the named service is not responding and that Bonjour's cache data for the service may be stale. Listing 4-5 (page 32) shows an example of how to use the `DNSServiceReconfirmRecord` function.

Following these rules helps Bonjour remove stale services from browse lists faster.

Listing 4-5 Example of calling DNSServiceReconfirmRecord

```
            // serviceName should be in the form
            // "name.service.protocol.domain.".  For example:
            // "MyLaptop._ftp._tcp.local."
            NSString *serviceName = ...;
            NSString *              serviceName;
            NSArray *               serviceNameComponents;
            NSUInteger              serviceNameComponentsCount;
          serviceNameComponents = [serviceName componentsSeparatedByString:@"."];
            serviceNameComponentsCount = [serviceNameComponents count];
            if ( (serviceNameComponentsCount >= 5) &&
([serviceNameComponents[serviceNameComponentsCount - 1] length] == 0) ) {
                protocol = [serviceNameComponents[2] lowercaseString];
                if ( [protocol isEqual:@"_tcp"] || [protocol isEqual:@"_udp"] ) {
                    fullname = [[serviceNameComponents
subarrayWithRange:NSMakeRange(1, serviceNameComponentsCount - 1)]
componentsJoinedByString:@"."];
                    retVal = EXIT_SUCCESS;
                    NSMutableData *     recordData;

                    recordData = [[NSMutableData alloc] init];
                    for (NSString * label in serviceNameComponents) {
                        const char *    labelStr;
                        uint8_t         labelStrLen;

                        labelStr = [label UTF8String];
                        if (strlen(labelStr) >= 64) {
                            fprintf(stderr, "%s: label too long: %s\n",
getprogname(), labelStr);
                            retVal = EXIT_FAILURE;
                            break;
                        } else {
                            // cast is safe because of length check
                            labelStrLen = (uint8_t) strlen(labelStr);
```

```
                            [recordData appendBytes:&labelStrLen
length:sizeof(labelStrLen)];
                            [recordData appendBytes:labelStr length:labelStrLen];
                        }
                    }

                    if ( (retVal == EXIT_SUCCESS) && ([recordData length] >= 256)
 ) {
                     fprintf(stderr, "%s: record data too long\n", getprogname());
                        retVal = EXIT_FAILURE;
                    }

                    if (retVal == EXIT_SUCCESS) {
                        err = DNSServiceReconfirmRecord(
                            0,
                            interfaceIndex,
                            [fullname UTF8String],
                            kDNSServiceType_PTR,
                            kDNSServiceClass_IN,
                            // cast is safe because of recordData length check
above
                            (uint16_t) [recordData length],
                            [recordData bytes]
                        );
                        if (err != kDNSServiceErr_NoError) {
                            fprintf(stderr, "%s: reconfirm record error: %d\n",
getprogname(), (int) err);
                            retVal = EXIT_FAILURE;
                        }
                    }
                }
            }
```

Connecting to a Bonjour Service by IP Address

As a rule, you should not resolve a service to an IP address and port number unless you are doing something very unusual. In the dynamic world of modern networking, IP addresses can change at any time. Further, a given Bonjour service can have more than one IP address, and not all of them may be reachable. By connecting using hostnames, you avoid these problems. For more information, read "Avoid Resolving DNS Names Before Connecting to a Host" in *Networking Overview*.

In the rare instances when connecting by IP address is required, you can do so by following the steps for resolving a Bonjour service in "Connecting by Name" (page 26) and then using the IP addresses instead of the hostname. There are a few caveats, however:

- If you are working with IP addresses directly, be sure to perform resolution every time you use a service, because although the service name is a persistent property, the socket information (IP addresses and port number) can change from session to session.

  In particular, never store the IP address information long-term. If a user browses for a service and saves that service, such as in a printer chooser, store only the service name, type, and domain. When it is time to connect, these values can be resolved into the relevant IP address and port number. See "Persisting a Bonjour Service" (page 34) for more information.

- The `netServiceDidResolveAddress:` method tells the delegate that the `NSNetService` object has added an address to its list of addresses for the service. However, more addresses may be added. For example, in systems that support both IPv4 and IPv6, `netServiceDidResolveAddress:` may be called two or more times—once for the IPv4 address and again for the IPv6 address.

  In this delegate method, be sure that all the required connection information is present before attempting to connect to the service. If some of the information is missing, `netServiceDidResolveAddress:` will be called later. If multiple addresses are returned from resolving a service, try to connect to each one before giving up.

Persisting a Bonjour Service

If your app needs to persistently store a reference to a Bonjour service, such as in a printer chooser, store only the service name, type, and domain. By persisting only the domain, type, and name information, you ensure that your app can find the relevant service even if its IP addresses or port number has changed.

When you connect to the service later, initialize an `NSNetService` object with this information by calling its `initWithDomain:type:name:` method.

After you have initialized the service object, connect to the service by calling the `getInputStream:outputStream:` method or by passing the result of a `hostName` call to a connect-by-name function or method, as described earlier in this chapter. When you do this, Bonjour automatically resolves the hostname or domain, type, and name values into the current IP addresses and port number for the service.

Monitoring a Bonjour Service

In some situations, an app may need to know certain information about a Bonjour service without needing to maintain a connection to that service. For example, a chat program might let the user specify a status (idle, away, and so on). Other users on the network should be able to learn this new information quickly, without each machine needing to constantly poll every other machine.

To support the dissemination of such information, Bonjour lets each service provide an arbitrary DNS `TXT` record with additional information. If you want to use this functionality in your app, you must do the following:

When publishing a service, call the `setTXTRecordData:` method to set or update the data associated with the service. This data must be encoded according to section 3.3.14 of RFC 1035.

The most straightforward way to generate compliant data is by calling `dataFromTXTRecordDictionary:` with an `NSDictionary` object. In that dictionary, store *only* the key-value pairs that you want to make available to any app that is browsing for your service.

Important: There are a few rules you must follow when using the `dataFromTXTRecordDictionary:` method:

- Each key must be no longer than 255 bytes in length.
- Each value must be no longer than 255 bytes in length.
- The total size of the `TXT` record, including length bytes for each key and value in the dictionary, *must* be no longer than 65,535 bytes.

  In addition, for performance reasons, it is strongly recommended that you limit your `TXT` records so that they fit inside a single Ethernet packet (about 1300 bytes).

If the data you need to provide is too large to fit, you can either split long values across multiple keys or use the dictionary to store a monotonically increasing version number so that interested clients know to open a connection to the service and request the actual data.

When browsing for services, do the following:

- Call the `startMonitoring` method to start monitoring the `TXT` record associated with a given service.

- Implement the `netService:didUpdateTXTRecordData:` method to receive notification whenever a service's `TXT` record changes.

  You can either process the resulting data yourself or call `dictionaryFromTXTRecordData:` to obtain an `NSDictionary` object containing the provided key-value pairs (often from a previous call to `dataFromTXTRecordDictionary:`).
- Call the `stopMonitoring` method when you no longer want to receive notifications about changes to a particular service.

For more details, see the documentation for the methods mentioned above.

Using CFNetServices

`CFNetServices` is an API in the `CFNetwork` framework (Core Foundation level) that allows you to publish or search for network services.

> **Important:** Using the `CFNetServices` API is generally discouraged unless you are writing code based on Core Foundation and have a strict policy of avoiding Objective-C. For all other software, the Foundation-based NSNetServices API or the DNS Service Discovery C API is generally a better choice. If you want a high-level API, use NSNetServices. If you want a C API, use DNS Service Discovery.

At a high level, the `CFNetServices` API provides access to Bonjour through three objects:

- `CFNetService`—An object that represents a single service on the network. A `CFNetService` object has a name, a type, a domain, and a port number. Service types used by `CFNetServices` are maintained at http://www.dns-sd.org/servicetypes.html.
- `CFNetServiceBrowser`—An object used to discover domains and discover network services within domains.
- `CFNetServiceMonitor`—An object used to monitor services for changes to their TXT records.

Publishing a Service Using CFNetServices

Publishing a service on the network involves two tasks: creating a service and registering a service. The next two sections describe what is required to perform these two tasks.

Creating a CFNetService Object

To create a `CFNetService` object, call `CFNetServiceCreate` and provide the following information about the service:

- **Name.** The human-readable name of the service (such as "`Sales Laser Printer`")
- **Service Type.** The type of service, such as "`_printer._tcp`";
- **Domain.** The domain for the service, typically the empty string (`CFSTR("")`)for default domain(s), or `local.` for the local domain only
- **Port.** The port number the service listens on

> **Note:** The dot in "`local.`" is part of the domain name. It signifies that the domain is fully qualified, which prevents anything from being added to the end of the domain (for example, `local.com`).

If you are implementing a protocol that relies on data stored in DNS text records, you can associate that information with a `CFNetService` object by calling `CFNetServiceSetTXTData`).

Associate a callback function with your `CFNetService` object by calling `CFNetServiceSetClient`. Your callback function is called to report errors that occur while your service is running and to report on the status of publication.

If you want the service to run asynchronously, you must also schedule the service on a run loop by calling `CFNetServiceScheduleWithRunLoop`; otherwise, the service will run synchronously. For more information about the modes in which a service can run, see"Asynchronous and Synchronous Modes" (page 43).

Listing A-1 shows how to create a `CFNetService` object.

Listing A-1 Creating a `CFNetService` object

```
    CFStringRef serviceType = CFSTR("_printer._tcp");
    CFStringRef serviceName = CFSTR("Sales Laser Printer");
    CFStringRef theDomain = CFSTR("");
    int chosenPort = 515;


    CFNetServiceRef netService = CFNetServiceCreate(NULL, theDomain, serviceType,
 serviceName, chosenPort);
```

Registering a CFNetService

To make a service available on the network, call `CFNetServiceRegisterWithOptions`. (This operation is also known as "publishing" a service.) The service can then be found by clients until you unregister the service by calling `CFNetServiceCancel`.

See Listing A-2 for sample code on this subject.

Listing A-2 Registering an asynchronous service

```
void registerCallback (
   CFNetServiceRef theService,
   CFStreamError* error,
```

```
    void* info)
{
    // ...
}

void startBonjour (CFNetServiceRef netService) {
    CFStreamError error;
    CFNetServiceClientContext clientContext = { 0, NULL, NULL, NULL, NULL };

    CFNetServiceSetClient(netService, registerCallback, &clientContext);
    CFNetServiceScheduleWithRunLoop(netService, CFRunLoopGetCurrent(),
kCFRunLoopCommonModes);
    CFOptionFlags options = 0; // or kCFNetServiceFlagNoAutoRename

    if (CFNetServiceRegisterWithOptions(netService, options, &error) == false) {
      CFNetServiceUnscheduleFromRunLoop(netService,
CFRunLoopGetCurrent(),kCFRunLoopCommonModes);
      CFNetServiceSetClient(netService, NULL, NULL);
      CFRelease(netService);
      fprintf(stderr, "could not register Bonjour service");
    }
}
```

Browsing for Services using CFNetServices

To browse for services represented by a `CFNetService` object, call `CFNetServiceBrowserCreate` and provide a pointer to a callback function that will be called as services are found.

If you want searches to be conducted asynchronously, you must also schedule the browser on a run loop by calling `CFNetServiceBrowserScheduleWithRunLoop`.

To browse for services, you can call `CFNetServiceBrowserSearchForServices` and specify the services to search for. For the domain parameter, you have two options. It is recommended that you pass the empty string (`CFSTR("")`) as the domain, allowing you to discover services in any domain on which your system is registered. Alternatively, you can specify a domain to search in. Your callback function will be called and passed a `CFNetService` object representing a matching service. The `CFNetServiceBrowser` object continues searching until your app stops the search by calling `CFNetServiceBrowserStopSearch`.

For each `CFNetService` object that your callback function receives, you can call `CFNetServiceResolveWithTimeout` to update the service with the IP address for the service. Then call `CFNetServiceGetAddressing` to get an array of `CFDataRef` objects, one for each IP address associated with the service. Each `CFDataRef` object, in turn, contains a `sockaddr` structure with an IP address.

A good example of how to browse for services can be seen in Listing A-3.

Listing A-3 Browsing asynchronously for services

```
void MyBrowseCallBack (
   CFNetServiceBrowserRef browser,
   CFOptionFlags flags,
   CFTypeRef domainOrService,
   CFStreamError* error,
   void* info)
{
    // ...
}

static Boolean MyStartBrowsingForServices(CFStringRef type, CFStringRef domain) {
     CFNetServiceClientContext clientContext = { 0, NULL, NULL, NULL, NULL };
     CFStreamError error;
     Boolean result;

     assert(type != NULL);

     CFNetServiceBrowserRef gServiceBrowserRef =
CFNetServiceBrowserCreate(kCFAllocatorDefault, MyBrowseCallBack, &clientContext);
     assert(gServiceBrowserRef != NULL);

     CFNetServiceBrowserScheduleWithRunLoop(gServiceBrowserRef,
CFRunLoopGetCurrent(), kCFRunLoopCommonModes);

     result = CFNetServiceBrowserSearchForServices(gServiceBrowserRef, domain,
type, &error);
     if (result == false) {
```

```
        // Something went wrong, so let's clean up.
        CFNetServiceBrowserUnscheduleFromRunLoop(gServiceBrowserRef,
CFRunLoopGetCurrent(), kCFRunLoopCommonModes);       CFRelease(gServiceBrowserRef);
        gServiceBrowserRef = NULL;

        fprintf(stderr, "CFNetServiceBrowserSearchForServices returned (domain =
 %ld, error = %d)\n", (long)error.domain, error.error);
    }

    return result;
}
```

Resolving a Service Using CFNetServices

After you have a name, type, and domain, you can resolve the service to retrieve its hostname and port. As with registering a service, resolving a service also requires a `CFNetServiceRef` object. If you already have one (typically obtained through a browse operation callback), you don't need to take any further action. Otherwise, you can create one yourself by calling `CFNetServiceCreate`.

Important: If you create a `CFNetServiceRef` object yourself with the intent to use that object to resolve a service, you *must* pass a valid domain that was obtained when the service was first detected during browsing. Bonjour has no way to guess which domain you want to use when resolving the service name.

If you plan to resolve a service asynchronously, associate the newly created `CFNetServiceRef` object with a callback function, which will receive a `CFNetServiceRef` object and a pointer to a `CFStreamError` object. You can set this callback by calling `CFNetServiceSetClient`. Be sure to call `CFNetServiceScheduleWithRunLoop` afterward to add the service to a run loop (which is typically the result of a call to `CFRunLoopGetCurrent`).

After setting up the run loop, call `CFNetServiceResolve`. If this call returns an error, you should clean up all the references you created. Otherwise, just wait for your callback functions to be called.

An example of resolving a service with `CFNetService` is in Listing A-4.

Listing A-4 Resolving a service asynchronously

```
void MyResolveCallback (
   CFNetServiceRef theService,
```

```
   CFStreamError* error,
   void* info)
{
    // ...
}

static void MyResolveService(CFStringRef name, CFStringRef type, CFStringRef domain)
{
    CFNetServiceClientContext context = { 0, NULL, NULL, NULL, NULL };
    CFTimeInterval duration = 0; // use infinite timeout
    CFStreamError error;

   CFNetServiceRef gServiceBeingResolved = CFNetServiceCreate(kCFAllocatorDefault,
 domain, type, name, 0);
    assert(gServiceBeingResolved != NULL);

    CFNetServiceSetClient(gServiceBeingResolved, MyResolveCallback, &context);
    CFNetServiceScheduleWithRunLoop(gServiceBeingResolved, CFRunLoopGetCurrent(),
 kCFRunLoopCommonModes);

    if (CFNetServiceResolveWithTimeout(gServiceBeingResolved, duration, &error)
== false) {

        // Something went wrong, so let's clean up.
        CFNetServiceUnscheduleFromRunLoop(gServiceBeingResolved,
CFRunLoopGetCurrent(), kCFRunLoopCommonModes);
        CFNetServiceSetClient(gServiceBeingResolved, NULL, NULL);
        CFRelease(gServiceBeingResolved);
        gServiceBeingResolved = NULL;

        fprintf(stderr, "CFNetServiceResolve returned (domain = %ld, error =
%d)\n", (long)error.domain, error.error);
}

    return;
}
```

Monitoring a Service Using CFNetServices

`CFNetServiceMonitor` lets you watch services for changes to `TXT` records.

In the app that is publishing a service, you can provide a custom `TXT` record by calling `CFNetServiceSetTXTData`. The most straightforward way to provide spec-compliant data is by calling `CFNetServiceCreateTXTDataWithDictionary` and passing it a dictionary of values to publish.

In the app that needs to monitor a service, perform the following steps:

1. After you have a `CFNetServiceRef` object for the service you wish to monitor, create a monitor reference (`CFNetServiceMonitorRef`) by calling `CFNetServiceMonitorCreate`.
2. Schedule the monitor reference on a run loop with `CFNetServiceMonitorScheduleWithRunLoop`. (You can obtain the default run loop by calling `CFRunLoopGetCurrent`.)
3. Start the monitor by calling `CFNetServiceMonitorStart`, passing it the monitor reference and a callback function to handle the resulting data. Be sure to check the return value to ensure that the monitor started successfully.
4. In the callback function, the most straightforward way to handle the `TXT` data is by calling `CFNetServiceCreateDictionaryWithTXTData` to obtain a dictionary containing the original key-value pairs.

When you no longer need to monitor a service, call `CFNetServiceMonitorStop` (to stop the monitoring), `CFNetServiceMonitorUnscheduleFromRunLoop` (to unschedule your monitor from its run loop), and `CFNetServiceMonitorInvalidate` (to destroy the monitor reference).

Important: The length restrictions and other limitations described in "Monitoring a Bonjour Service" (page 35) also apply to this API.

Asynchronous and Synchronous Modes

Several `CFNetServices` functions can operate in asynchronous or synchronous mode. Scheduling a `CFNetService` or `CFNetServiceBrowser` object on a run loop causes the service or browser to operate in asynchronous mode. If a `CFNetService` or `CFNetServiceBrowser` object is not scheduled on a run loop, it operates in synchronous mode. Operating in asynchronous mode changes the behavior of its functions.

Although it is possible to use the synchronous modes of these functions, keep in mind that it is unacceptable to block the user interface or other functions of your program while you wait for synchronous functions to return. Because network operations may take an arbitrary amount of time to complete, it is highly recommended that you use the asynchronous modes of each function.

Table A-1 describes the differences in behavior between synchronous and asynchronous modes.

Table A-1 Behavior of certain `CFNetServices` functions in asynchronous and synchronous mode

Function	Asynchronous mode	Synchronous mode
`CFNetServiceRegister-WithOptions`	Starts the registration and returns. The callback function for the `CFNetService` is called to report any errors that occur while the service is running. The service is available on the network until your app cancels the registration.	Blocks until your app cancels the service from another thread or until an error occurs, at which point the function returns. If an error occurs, the error is returned through the provided error structure. The service is available on the network until your app cancels the registration or an error occurs.
`CFNetServiceResolve-WithTimeout`	Starts the resolution and returns. The callback function for the `CFNetService` is called to report any errors that occur during resolution. The resolution operation runs until the specified timeout is reached or, if the timeout was specified as zero, until it is canceled.	Blocks until at least one IP address is found for the service, an error occurs, the time specified as the timeout parameter is reached, or your app cancels the resolution, at which point the function returns. If an error occurs, the error is returned through the provided error structure. The resolution operation continues to run until your app cancels it or an error occurs.
`CFNetServiceBrowser-SearchForDomains`	Starts the search and returns. The callback function for the `CFNetServiceBrowser` is called for each domain that is found and when any error occurs while browsing. Browsing continues to run until your app stops the browsing.	Blocks until an error occurs or your app calls `CFNetServiceBrowser-StopSearch`, at which time the callback function for the `CFNetServiceBrowser` is called for each domain that was found. Any error is returned through the provided error structure. Browsing continues until your app stops the browsing operation.

Function	Asynchronous mode	Synchronous mode
`CFNetServiceBrowser-SearchForServices`	Starts the search and returns. The callback function for the `CFNetServiceBrowser` is called for each `CFNetService` that is found and when any error occurs while browsing. Browsing continues to run until your app stops the browsing.	Blocks until an error occurs or until your app calls `CFNetServiceBrowser-StopSearch`, at which time the callback function for the `CFNetServiceBrowser` is called for each `CFNetService` that was found. Any error is returned in the provided error structure. Browsing continues until your app stops the browsing.

Shutting Down Services and Searches

To shut down a service that is running in asynchronous mode, your app unschedules the service from all run loops that it may be scheduled on. The app then calls `CFNetServiceSetClient` with the `clientCB` parameter set to `NULL` to disassociate your callback function from the `CFNetService` object. Finally, the app calls `CFNetServiceCancel` to stop the service.

To shut down a service that is running in synchronous mode, your app only needs to call `CFNetServiceCancel` from another thread.

Listing A-5 shows a good example of shutting down an asynchronous `CFNetService` resolve operation that has not timed out.

Listing A-5 Canceling an asynchronous `CFNetService` resolve operation

```
void MyCancelResolve(CFNetServiceRef gServiceBeingResolved)
{
     assert(gServiceBeingResolved != NULL);
    CFNetServiceUnscheduleFromRunLoop(gServiceBeingResolved, CFRunLoopGetCurrent(),
 kCFRunLoopCommonModes);
     CFNetServiceSetClient(gServiceBeingResolved, NULL, NULL);
     CFNetServiceCancel(gServiceBeingResolved);
     CFRelease(gServiceBeingResolved);
     gServiceBeingResolved = NULL;
     return;
}
```

To shut down a browser that is running in asynchronous mode, your app unschedules the browser from all run loops that it may be scheduled on and then calls `CFNetServiceBrowserInvalidate`. Then your app calls `CFNetServiceBrowserStopSearch`. If the browser is running in synchronous mode, you only need to call `CFNetServiceBrowserStopSearch`. An example of these functions can be seen in Listing A-6.

Listing A-6 Stop browsing for services

```
static void MyStopBrowsingForServices(CFNetServiceBrowserRef gServiceBrowserRef)
{
     CFStreamError streamerror;
     assert(gServiceBrowserRef != NULL);
     CFNetServiceBrowserStopSearch(gServiceBrowserRef, &streamerror);
     CFNetServiceBrowserUnscheduleFromRunLoop(gServiceBrowserRef,
CFRunLoopGetCurrent(), kCFRunLoopCommonModes);
     CFNetServiceBrowserInvalidate(gServiceBrowserRef);
     CFRelease(gServiceBrowserRef);
     gServiceBrowserRef = NULL;
     return;
}
```

Browsing for Domains

Depending on your app, you may or may not need to browse for domains. You should consider browsing for domains if:

- You want to provide the ability for the user to control which domains are used for service-browsing purposes.
- You want to provide the ability for the user to control which domains your app uses when publishing its services.
- You are writing a specialized Bonjour browser and you want to show all possible domains in your service-browsing results, even if those domains did not return any results.
- You need to obtain a complete list of all domains that the computer knows about, even if they are not enabled by default.

If your app does not meet any of those criteria, you probably do not need to browse for domains. However, if your app lets the user browse for services, you should still design your user interface in such a way that the user can distinguish between multiple identical results provided by different domains.

This chapter explains how to discover what domains are available for use in future browsing or service registration activities.

> **Important:** Running dozens of browsers is expensive, and each browse request takes time to complete. For these reasons, you should not browse for domains and then browse within each domain (whether sequentially or in parallel). Instead, either browse for services in all domains at once or browse in the local domain initially and then present a list of domains so that the user can choose a different domain to browse.

About Domain Browsing

The `NSNetServiceBrowser` class provides methods for browsing available domains. Use these methods if you are writing an app that needs to publish or browse domains other than the default domains.

Because domain browsing can take time, `NSNetServiceBrowser` objects perform browsing asynchronously by registering with a run loop. Browsing results are returned to your app through delegate methods. To correctly use an `NSNetServiceBrowser` object, you must assign it a delegate.

Browsing for domains takes three steps:

1. Initialize an `NSNetServiceBrowser` instance, and assign a delegate to the object.
2. Begin a search for domains (either for registration or for browsing).
3. Handle search results and other messages sent to the delegate object.

The following sections describe these steps in detail.

Initializing the Browser and Starting a Search

To initialize an `NSNetServiceBrowser` object, use the `init` method. This method sets up the browser and adds it to the current run loop. If you want to use a run loop other than the current one, use the `removeFromRunLoop:forMode:` and `scheduleInRunLoop:forMode:` methods.

After you initialize the object, you can use it to search for domains in which you can register services (with the `searchForRegistrationDomains` method) or browse for services (with the `searchForBrowseDomains` method).

When you are finished, call the `stop` method. You should perform any necessary cleanup in the `netServiceBrowserDidStopSearch:` delegate callback.

Listing B-1 demonstrates how to browse for registration domains with `NSNetServiceBrowser`. The code initializes the object, assigns a delegate, and begins a search for available domains.

Listing B-1 Browsing for registration domains

```
id delegateObject; // Assume this object exists.
NSNetServiceBrowser *domainBrowser;

domainBrowser = [[NSNetServiceBrowser alloc] init];
[domainBrowser setDelegate:delegateObject];
[domainBrowser searchForRegistrationDomains];
```

Implementing Delegate Methods for Browsing

`NSNetServiceBrowser` returns all browsing results to its delegate. If you are using the class to browse for domains, your delegate object should implement the following methods:

`netServiceBrowserWillSearch:`

`netServiceBrowserDidStopSearch:`

`netServiceBrowser:didNotSearch:`

`netServiceBrowser:didFindDomain:moreComing:`

`netServiceBrowser:didRemoveDomain:moreComing:`

The `netServiceBrowserWillSearch:` method notifies the delegate that a search is commencing. You can use this method to update your user interface to reflect that a search is in progress. When browsing stops, the delegate receives a `netServiceBrowserDidStopSearch:` message. In that delegate method, you can perform any necessary cleanup.

If the delegate receives a `netServiceBrowser:didNotSearch:` message, it means that the search failed for some reason. You should extract the error information from the dictionary with the `NSNetServicesErrorCode` key and handle the error accordingly. See `NSNetServicesError` for a list of possible errors.

You track domains with two methods—`netServiceBrowser:didFindDomain:moreComing:` and `netServiceBrowser:didRemoveDomain:moreComing:`—which indicate that a service has become available or has shut down. The `moreComing` parameter indicates whether more results are on the way. If this parameter is `YES`, you should not update any user interface elements until the method is called with a `moreComing` parameter of `NO`. If you want a list of available domains, you need to maintain your own array based on the information provided by delegate methods.

> **Note:** As with browsing within a domain, a `moreComing:` value of `NO` does not mean that you will receive no further callbacks. It is an indication that your app has been provided with the complete state of the world as Bonjour understands it up to that point in time. There is still a chance that more domains will become available as new networks appear, new services appear on existing networks, or unusually slow devices respond to requests.

Listing B-2 shows the interface for a class that responds to the `NSNetServiceBrowser` delegate methods required for domain browsing, and Listing B-3 shows its implementation. You can use this code as a starting point for your domain browsing code.

Listing B-2 Interface for an `NSNetServiceBrowser` delegate object used when browsing for domains

```
#import <Foundation/Foundation.h>


@interface NetServiceDomainBrowserDelegate : NSObject <NSNetServiceBrowserDelegate>
```

```
{
    // Keeps track of available domains
    NSMutableArray *domains;

    // Keeps track of search status
    BOOL searching;
}

// NSNetServiceBrowser delegate methods for domain browsing
- (void)netServiceBrowserWillSearch:(NSNetServiceBrowser *)browser;
- (void)netServiceBrowserDidStopSearch:(NSNetServiceBrowser *)browser;
- (void)netServiceBrowser:(NSNetServiceBrowser *)browser
        didNotSearch:(NSDictionary *)errorDict;
- (void)netServiceBrowser:(NSNetServiceBrowser *)browser
        didFindDomain:(NSString *)domainString
        moreComing:(BOOL)moreComing;
- (void)netServiceBrowser:(NSNetServiceBrowser *)browser
        didRemoveDomain:(NSString *)domainString
        moreComing:(BOOL)moreComing;

// Other methods
- (void)handleError:(NSNumber *)error;
- (void)updateUI;

@end
```

Listing B-3 Implementation for an `NSNetServiceBrowser` delegate object used when browsing for domains

```
#import "NetServiceDomainBrowserDelegate.h"

@implementation NetServiceDomainBrowserDelegate

- (id)init
{
    self = [super init];
```

```
    if (self) {
        domains = [[NSMutableArray alloc] init];
        searching = NO;
    }
    return self;
}

// Sent when browsing begins
- (void)netServiceBrowserWillSearch:(NSNetServiceBrowser *)browser
{
    searching = YES;
    [self updateUI];
}

// Sent when browsing stops
- (void)netServiceBrowserDidStopSearch:(NSNetServiceBrowser *)browser
{
    searching = NO;
    [self updateUI];
}

// Sent if browsing fails
- (void)netServiceBrowser:(NSNetServiceBrowser *)browser
        didNotSearch:(NSDictionary *)errorDict
{
    searching = NO;
    [self handleError:[errorDict objectForKey:NSNetServicesErrorCode]];
}

// Sent when a domain appears
- (void)netServiceBrowser:(NSNetServiceBrowser *)browser
        didFindDomain:(NSString *)domainString
        moreComing:(BOOL)moreComing
{
    [domains addObject:domainString];
```

```
    if(!moreComing)
    {
        [self updateUI];
    }
}

// Sent when a domain disappears
- (void)netServiceBrowser:(NSNetServiceBrowser *)browser
        didRemoveDomain:(NSString *)domainString
        moreComing:(BOOL)moreComing
{
    [domains removeObject:domainString];

    if(!moreComing)
    {
        [self updateUI];
    }
}

// Error handling code
- (void)handleError:(NSNumber *)error
{
    NSLog(@"An error occurred. Error code = %@", error);
    // Handle error here
}

// UI update code
- (void)updateUI
{
    if(searching)
    {
        // Update the user interface to indicate searching
        // Also update any UI that lists available domains
    }
```

```
    else
    {
        // Update the user interface to indicate not searching
    }
}

@end
```

Document Revision History

This table describes the changes to *NSNetServices and CFNetServices Programming Guide*.

Date	Notes
2013-04-23	Updated code listings for ARC, fixed various technical errors, and updated artwork.
2012-06-11	Made minor change.
2010-03-24	Updated code examples to new initialization pattern.
2009-10-09	Corrected code listings.
2008-10-15	Updated code listings.
2005-11-09	Removed Preliminary stamp.
2005-10-04	New document that describes how to implement Bonjour in Cocoa or Carbon applications.

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