CFNetwork Programming Guide

Networking

Contents

Figures and Listings

Introduction to CFNetwork Programming Guide

CFNetwork is a framework in the Core Services framework that provides a library of abstractions for network protocols. These abstractions make it easy to perform a variety of network tasks, such as:

- Working with BSD sockets
- Creating encrypted connections using SSL or TLS
- Resolving DNS hosts
- Working with HTTP, authenticating HTTP and HTTPS servers
- Working with FTP servers
- Publishing, resolving and browsing Bonjour services

This book is intended for developers who want to use network protocols in their applications. In order to fully understand this book, the reader should have a good understanding of network programming concepts such as BSD sockets, streams and HTTP protocols. Additionally, the reader should be familiar Mac OS X programming concepts including run loops. For more information about Mac OS X please read *Mac OS X Technology Overview*.

Organization of This Document

This book contains the following chapters:

- "CFNetwork Concepts" (page 9) describes each of the CFNetwork APIs and how they interact.
- "Working with Streams" (page 17) describes how to use the CFStream API to send and receive network data.
- "Communicating with HTTP Servers" (page 27) describes how to send and receive HTTP messages.
- "Communicating with Authenticating HTTP Servers" (page 33) describes how to communicate with secure HTTP servers.
- "Working with FTP Servers" (page 43) describes how to upload and download files from an FTP server, and how to download directory listings.
- "Using Network Diagnostics" (page 49) describes how to add network diagnostics to your application.

See Also

For more information about the networking APIs in Mac OS X, read:

- *Getting Started with Networking*

Refer to the following reference documents for CFNetwork:

- *CFFTPStream Reference* is the reference documentation for the CFFTPStream API.
- *CFHTTPMessage Reference* is the reference documentation for the CFHTTPMessage API.
- *CFHTTPStream Reference* is the reference documentation for the CFHTTPStream API.
- *CFHTTPAuthentication Reference* is the reference documentation for the CFHTTPAuthentication API.
- *CFHost Reference* is the reference documentation for the CFHost API.
- *CFNetServices Reference* is the reference documentation for the CFNetServices API.
- *CFNetDiagnostics Reference* is the reference documentation for the CFNetDiagnostics API.

In addition to the documentation provided by Apple, the following is the reference book for socket-level programming:

- *UNIX Network Programming, Volume 1* (Stevens, Fenner and Rudoff)

CFNetwork Concepts

CFNetwork is a low-level, high-performance framework that gives you the ability to have detailed control over the protocol stack. It is an extension to BSD sockets, the standard socket abstraction API that provides objects to simplify tasks such as communicating with FTP and HTTP servers or resolving DNS hosts. CFNetwork is based, both physically and theoretically, on BSD sockets.

Just as CFNetwork relies on BSD sockets, there are a number of Cocoa classes that rely on CFNetwork. `NSURL` is one such class used for communicating with servers using standard Internet protocols. In addition, the Web Kit is a set of Cocoa classes to display web content in windows. Both of these classes are very high level and implement most of the details of the networking protocols by themselves. Thus, the structure of the software layers looks like the image in Figure 1-1.

Figure 1-1   CFNetwork and other software layers on Mac OS X

| Web Kit |
| NSURL |
| CFNetwork |
| BSD sockets |
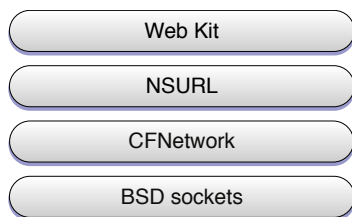
When to Use CFNetwork

CFNetwork has a number of advantages over BSD sockets. It provides run-loop integration, so if your application is run loop based you can use network protocols without implementing threads. CFNetwork also contains a number of objects to help you use network protocols without having to implement the details yourself. For example, you can use FTP protocols without having to implement all of the details with the CFFTP API. If you understand the networking protocols and need the low-level control they provide but don't want to implement them yourself, then CFNetwork is probably the right choice.

There are a number of advantages of using CFNetwork instead of the Cocoa framework `NSURL`. CFNetwork is focused more on the network protocols, whereas `NSURL` is focused more on data access, such as transferring data over HTTP or FTP. Although `NSURL` does provide some configurability, CFNetwork provides a lot more. Additionally, `NSURL` requires that you use Objective-C. If that is not feasible, then you should use CFNetwork. For more information on the Foundation networking frameworks, read *URL Loading System*.

Now that you understand how CFNetwork interacts with the other Mac OS X networking APIs, you're ready to become familiar with the CFNetwork APIs along with two APIs that form the infrastructure for CFNetwork.

CFNetwork Infrastructure

Before learning about the CFNetwork APIs, you must first understand the APIs which are the foundation for the majority of CFNetwork. CFNetwork relies on two APIs that are part of the Core Foundation framework, CFSocket and CFStream. Understanding these APIs is essential to using CFNetwork.

CFSocket API

Sockets are the most basic level of network communications. A socket acts in a similar manner to a telephone jack. It allows you to connect to another socket (either locally or over a network) and send data to that socket.

The most common socket abstraction is BSD sockets. CFSocket is an abstraction for BSD sockets. With very little overhead, CFSocket provides almost all the functionality of BSD sockets, and it integrates the socket into a run loop. CFSocket is not limited to stream-based sockets (for example, TCP), it can handle any type of socket.

You could create a CFSocket object from scratch using the `CFSocketCreate` function, or from a BSD socket using the `CFSocketCreateWithNative` function. Then, you could create a run-loop source using the function `CFSocketCreateRunLoopSource` and add it to a run loop with the function `CFRunLoopAddSource`. This would allow your CFSocket callback function to be run whenever the CFSocket object receives a message.

Read *CFSocket Reference* for more information about the CFSocket API.

CFStream API

Read and write streams provide an easy way to exchange data to and from a variety of media in a device-independent way. You can create streams for data located in memory, in a file, or on a network (using sockets), and you can use streams without loading all of the data into memory at once.

A stream is a sequence of bytes transmitted serially over a communications path. Streams are one-way paths, so to communicate bidirectionally an input (read) stream and output (write) stream are necessary. Except for ones that are file based, streams are nonseekable; after stream data has been provided or consumed, it cannot be retrieved again from the stream.

CFStream is an API that provides an abstraction for these streams with two new CFType objects: CFReadStream and CFWriteStream. Both types of stream follow all of the usual Core Foundation API conventions. For more information about Core Foundation types, read *Core Foundation Design Concepts*.

CFStream is built on top of CFSocket and is the foundation for CFHTTP and CFFTP. As you can see in Figure 1-2, even though CFStream is not officially part of CFNetwork, it is the basis for almost all of CFNetwork.

Figure 1-2   CFStream API structure
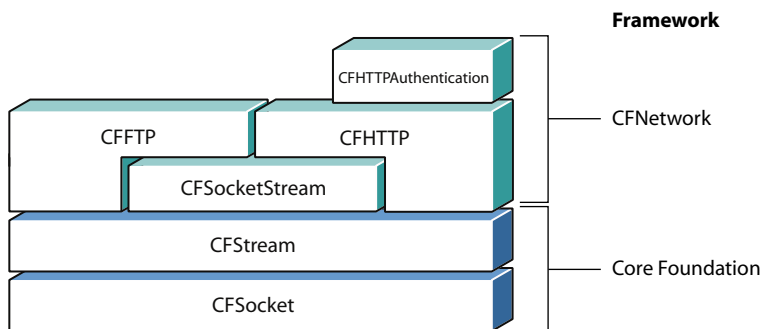
You can use read and write streams in much the same way as you do UNIX file descriptors. First, you instantiate the stream by specifying the stream type (memory, file, or socket) and set any options. Next, you open the stream and read or write any number of times. While the stream exists, you can get information about the stream by asking for its properties. A stream property is any information about the stream, such as its source or destination, that is not part of the actual data being read or written. When you no longer need the stream, close and dispose of it.

CFStream functions that read or write a stream will suspend, or *block*, the current process until at least one byte of the data can be read or written. To avoid trying to read from or write to a stream when the stream would block, use the asynchronous version of the functions and schedule the stream on a run loop. Your callback function is called when it is possible to read and write without blocking.

In addition, CFStream has built-in support for the Secure Sockets Layer (SSL) protocol. You can set up a dictionary containing the stream's SSL information, such as the security level desired or self-signed certificates. Then pass it to your stream as the `kCFStreamPropertySSLSettings` property to make the stream an SSL stream.

It is not possible to construct a custom flavor of CFStream. For example, if you want to stream data from an object embedded within a custom database file, you can't do this by creating your own CFStream flavor. Instead, you implement a custom subclass of NSStream (using Objective-C). Since NSStream is toll free bridged to CFStream, your NSStream subclass can be used wherever a CFStream is required. For more information about the NSStream classes, read *Stream Programming Guide for Cocoa*.

The chapter "Working with Streams" (page 17) describes how to use read and write streams.

CFNetwork API Concepts

To understand the CFNetwork framework, you need to be familiar with the building blocks that compose it. The CFNetwork framework is broken up into separate APIs, each covering a specific network protocol. These APIs can be used in combination, or separately, depending on your application. Most of the programming conventions are common among the APIs, so it's important to comprehend each of them.

CFFTP API

Communicating with an FTP server is made easier with CFFTP. Using the CFFTP API, you can create FTP read streams (for downloading) and FTP write streams (for uploading). Using FTP read and write streams you can perform functions such as:

- Download a file from an FTP server

- Upload a file to an FTP server

- Download a directory listing from an FTP server

- Create directories on an FTP server

An FTP stream works like all other CFNetwork streams. For example, you can create an FTP read stream by calling the function `CFReadStreamCreateWithFTPURL` function. Then, you can call the function `CFReadStreamGetError` at any time to check the status of the stream.

By setting properties on FTP streams, you can adapt your stream for its particular application. For example, if the server that the stream is connecting to requires a user name and password, you need to set the appropriate properties so the stream can work properly. For more information about the different properties available to FTP streams read "Setting up the Streams" (page 43).

A CFFTP stream can be used synchronously or asynchronously. To open the connection with the FTP server that was specified when the FTP read stream was created, call the function `CFReadStreamOpen`. To read from the stream, use the `CFReadStreamRead` function and provide the read stream reference, `CFReadStreamRef`, that was returned when the FTP read stream was created. The `CFReadStreamRead` function fills a buffer with the output from the FTP server.

For more information on using CFFTP, see "Working with FTP Servers" (page 43).

CFHTTP API

To send and receive HTTP messages, use the CFHTTP API. Just as CFFTP is an abstraction for FTP protocols, CFHTTP is an abstraction for HTTP protocols.

Hypertext Transfer Protocol (HTTP) is a request/response protocol between a client and a server. The client creates a request message. This message is then serialized, a process that converts the message into a raw byte stream. Messages cannot be transmitted until they are serialized first. Then the request message is sent to the server. The request typically asks for a file, such as a webpage. The server responds, sending back a string followed by a message. This process is repeated as many times as is necessary.

To create an HTTP request message, you specify the following:

- The request method, which can be one of the request methods defined by the Hypertext Transfer Protocol, such as `OPTIONS`, `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `TRACE`, and `CONNECT`

- The URL, such as `http://www.apple.com`

- The HTTP version, such as version 1.0 or 1.1

- The message's headers, by specifying the header name, such as `User-Agent`, and its value, such as `MyUserAgent`

- The message's body

After the message has been constructed, you serialize it. Following serialization, a request might look like this:

```
GET / HTTP/1.0\r\nUser-Agent: UserAgent\r\nContent-Length: 0\r\n\r\n
```

Deserialization is the opposite of serialization. With deserialization, a raw byte stream received from a client or server is restored to its native representation. CFNetwork provides all of the functions needed to get the message type (request or response), HTTP version, URL, headers, and body from an incoming, serialized message.

More examples of using CFHTTP are available in "Communicating with HTTP Servers" (page 27).

CFHTTPAuthentication API

If you send an HTTP request to an authentication server without credentials (or with incorrect credentials), the server returns an authorization challenge (more commonly known as a 401 or 407 response). The CFHTTPAuthentication API applies authentication credentials to challenged HTTP messages. CFHTTPAuthentication supports the following authentication schemes:

- Basic

- Digest

- NT LAN Manager (NTLM)

- Simple and Protected GSS-API Negotiation Mechanism (SPNEGO)

New in Mac OS X v10.4 is the ability to carry persistency across requests. In Mac OS X v10.3 each time a request was challenged, you had to start the authentication dialog from scratch. Now, you maintain a set of CFHTTPAuthentication objects for each server. When you receive a 401 or 407 response, you find the correct object and credentials for that server and apply them. CFNetwork uses the information stored in that object to process the request as efficiently as possible.

By carrying persistency across request, this new version of CFHTTPAuthentication provides much better performance. More information about how to use CFHTTPAuthentication is available in "Communicating with Authenticating HTTP Servers" (page 33).

CFHost API

You use the CFHost API to acquire host information, including names, addresses, and reachability information. The process of acquiring information about a host is known as *resolution*.

CFHost is used just like CFStream:

- Create a CFHost object.

- Start resolving the CFHost object.

- Retrieve either the addresses, host names, or reachability information.

- Destroy the CFHost object when you are done with it.

Like all of CFNetwork, CFHost is IPv4 and IPv6 compatible. Using CFHost, you could write code that handles IPv4 and IPv6 completely transparently.

CFHost is integrated closely with the rest of CFNetwork. For example, there is a CFStream function called `CFStreamCreatePairWithSocketToCFHost` that will create a CFStream object directly from a CFHost object. For more information about the CFHost object functions, see *CFHost Reference*.

CFNetServices API

If you want your application to use Bonjour to register a service or to discover services, use the CFNetServices API. Bonjour is Apple's implementation of zero-configuration networking (ZEROCONF), which allows you to publish, discover, and resolve network services.

To implement Bonjour the CFNetServices API defines three object types: CFNetService, CFNetServiceBrowser, and CFNetServiceMonitor. A CFNetService object represents a single network service, such as a printer or a file server. It contains all the information needed for another computer to resolve that server, such as name, type, domain and port number. A CFNetServiceBrowser is an object used to discover domains and network services within domains. And a CFNetServiceMonitor object is used to monitor a CFNetService object for changes, such as a status message in iChat.

For a full description of Bonjour, see *Bonjour Overview*. For more information about using CFNetServices and implementing Bonjour, see *NSNetServices and CFNetServices Programming Guide*.

CFNetDiagnostics API

Applications that connect to networks depend on a stable connection. If the network goes down, this causes problems with the application. By adopting the CFNetDiagnostics API, the user can self-diagnose network issues such as:

- Physical connection failures (for example, a cable is unplugged)

- Network failures (for example, DNS or DHCP server no longer responds)

- Configuration failures (for example, the proxy configuration is incorrect)

Once the network failure has been diagnosed, CFNetDiagnostics guides the user to fix the problem. You may have seen CFNetDiagnostics in action if Safari failed to connect to a website. The CFNetDiagnostics assistant can be seen in Figure 1-3.

Figure 1-3   Network diagnostics assistant
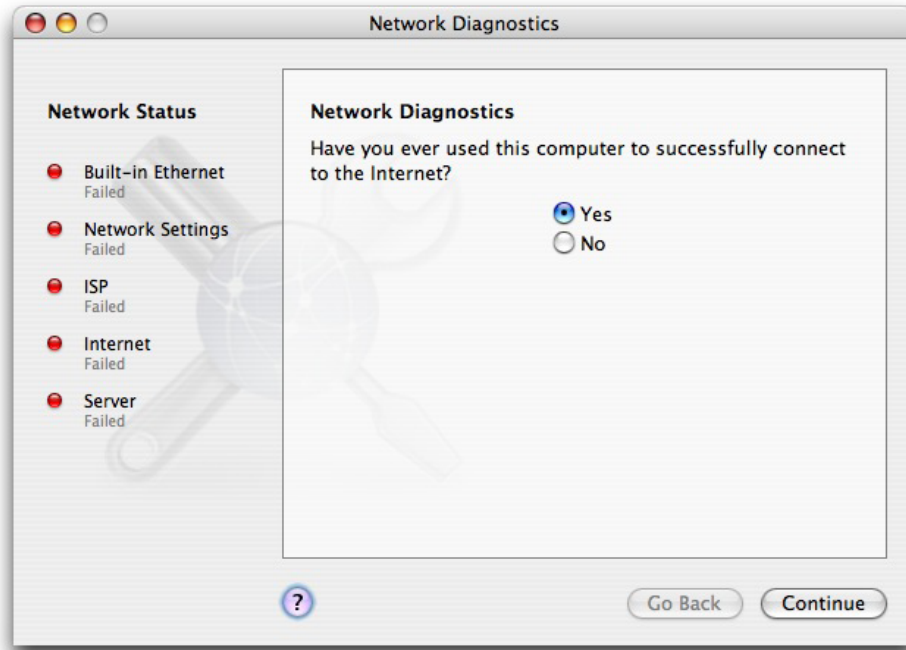
By providing CFNetDiagnostics with the context of the network failure, you can call the `CFNetDiagnosticDiagnoseProblemInteractively` function to lead the user through the prompts to find a solution. Additionally, you can use CFNetDiagnostics to query for connectivity status and provide uniform error messages to the user.

To see how to integrate CFNetDiagnotics into your application read "Using Network Diagnostics" (page 49). CFNetDiagnostics is a new API in Mac OS X v10.4.

Working with Streams

This chapter discusses how to create, open, and check for errors on read and write streams. It also describes how to read from a read stream, how to write to a write stream, how to prevent blocking when reading from or writing to a stream, and how to navigate a stream through a proxy server.

Working with Read Streams

Start by creating a read stream. Listing 2-1 creates a read stream for a file.

Listing 2-1   Creating a read stream from a file

```
CFReadStreamRef myReadStream = CFReadStreamCreateWithFile(kCFAllocatorDefault,
  fileURL);
```

In this listing, the `kCFAllocatorDefault` parameter specifies that the current default system allocator be used to allocate memory for the stream and the `fileURL` parameter specifies the name of the file for which this read stream is being created, such as `file:///Users/joeuser/Downloads/MyApp.sit`.

Now that the stream has been created, it can be opened. Opening a stream causes the stream to reserve any system resources that it requires, such as the file descriptor needed to open the file. Listing 2-2 is an example of opening the read stream.

Listing 2-2   Opening a read stream

```
if (!CFReadStreamOpen(myReadStream)) {
    CFStreamError myErr = CFReadStreamGetError(myReadStream);
    // An error has occurred.
        if (myErr.domain == kCFStreamErrorDomainPOSIX) {
        // Interpret myErr.error as a UNIX errno.
        } else if (myErr.domain == kCFStreamErrorDomainMacOSStatus) {
        // Interpret myErr.error as a MacOS error code.
            OSStatus macError = (OSStatus)myErr.error;
        // Check other error domains.
    }
}
```

The `CFReadStreamOpen` function returns `TRUE` to indicate success and `FALSE` if the open fails for any reason. If `CFReadStreamOpen` returns `FALSE`, the example calls the `CFReadStreamGetError` function, which returns a structure of type `CFStreamError` consisting of two values: a domain code and an error code. The domain code indicates how the error code should be interpreted. For example, if the domain code is `kCFStreamErrorDomainPOSIX`, the error code is a UNIX `errno` value. The other error domains are `kCFStreamErrorDomainMacOSStatus`, which indicates that the error code is an `OSStatus` value defined in `MacErrors.h`, and `kCFStreamErrorDomainHTTP`, which indicates that the error code is the one of the values defined by the `CFStreamErrorHTTP` enumeration.

Opening a stream can be a lengthy process, so the `CFReadStreamOpen` and `CFWriteStreamOpen` functions avoid blocking by returning `TRUE` to indicate that the process of opening the stream has begun. To check the status of the open, call the functions `CFReadStreamGetStatus` and `CFWriteStreamGetStatus`, which return `kCFStreamStatusOpening` if the open is still in progress, `kCFStreamStatusOpen` if the open is complete, or `kCFStreamStatusErrorOccurred` if the open has completed but failed. In most cases, it doesn't matter whether the open is complete because the CFStream functions that read and write will block until the stream is open.

To read from a read stream, call the function `CFReadStreamRead`, which is similar to the UNIX `read()` system call. Both take buffer and buffer length parameters. Both return the number of bytes read, `0` if at the end of stream or file, or `-1` if an error occurred. Both block until at least one byte can be read, and both continue reading as long as they can do so without blocking. Listing 2-3 is an example of reading from the read stream.

Listing 2-3   Reading from a read stream (blocking)

```
CFIndex numBytesRead;
do {
    UInt8 buf[kReadBufSize];
    numBytesRead = CFReadStreamRead(myReadStream, buf, sizeof(buf));
    if( numBytesRead > 0 ) {
        handleBytes(buf, numBytesRead);
    } else if( numBytesRead < 0 ) {
        CFStreamError error = CFReadStreamGetError(myReadStream);
        reportError(error);
    }
} while( numBytesRead > 0 );
```

When all data has been read, you should call the `CFReadStreamClose` function to close the stream, thereby releasing system resources associated with it. Then release the stream reference by calling the function `CFRelease`. You may also want to invalidate the reference by setting it to `NULL`. See Listing 2-4 for an example.

Listing 2-4   Releasing a read stream

```
CFReadStreamClose(myReadStream);
CFRelease(myReadStream);
myReadStream = NULL;
```

Working with Write Streams

Working with write streams is similar to working with read streams. One major difference is that the function `CFWriteStreamWrite` does not guarantee to accept all of the bytes that you pass it. Instead, `CFWriteStreamWrite` returns the number of bytes that it accepted. You'll notice in the sample code shown in Listing 2-5 that if the number of bytes written is not the same as the total number of bytes to be written, the buffer is adjusted to accommodate this.

Listing 2-5   Creating, opening, writing to, and releasing a write stream

```
CFWriteStreamRef myWriteStream =
        CFWriteStreamCreateWithFile(kCFAllocatorDefault, fileURL);
if (!CFWriteStreamOpen(myWriteStream)) {
```

```
    CFStreamError myErr = CFWriteStreamGetError(myWriteStream);
    // An error has occurred.
    if (myErr.domain == kCFStreamErrorDomainPOSIX) {
    // Interpret myErr.error as a UNIX errno.
    } else if (myErr.domain == kCFStreamErrorDomainMacOSStatus) {
        // Interpret myErr.error as a MacOS error code.
        OSStatus macError = (OSStatus)myErr.error;
        // Check other error domains.
    }
}
UInt8 buf[] = "Hello, world";
UInt32 bufLen = strlen(buf);

while (!done) {
    CFTypeRef bytesWritten = CFWriteStreamWrite(myWriteStream, buf, strlen(buf));
    if (bytesWritten < 0) {
        CFStreamError error = CFWriteStreamGetError(myWriteStream);
        reportError(error);
    } else if (bytesWritten == 0) {
        if (CFWriteStreamGetStatus(myWriteStream) == kCFStreamStatusAtEnd) {
            done = TRUE;
        }
    } else if (bytesWritten != strlen(buf)) {
        // Determine how much has been written and adjust the buffer
        bufLen = bufLen - bytesWritten;
        memmove(buf, buf + bytesWritten, bufLen);

        // Figure out what went wrong with the write stream
        CFStreamError error = CFWriteStreamGetError(myWriteStream);
        reportError(error);

    }
}
CFWriteStreamClose(myWriteStream);
CFRelease(myWriteStream);
myWriteStream = NULL;
```

Preventing Blocking When Working with Streams

When using streams to communicate, there is always a chance, especially with socket-based streams, that a data transfer could take a long time. If you are implementing your streams synchronously your entire application will be forced to wait on the data transfer. Therefore, it is highly recommended that your code use alternate methods to prevent blocking.

There are two ways to prevent blocking when reading from or writing to a CFStream object:

■ Polling — For read streams, find out if there are bytes to read before reading from the stream. For write streams, find out whether the stream can be written to without blocking before writing to the stream.

■ Using a run loop — Register to receive stream-related events and schedule the stream on a run loop. When a stream-related event occurs, your callback function (specified by the registration call) is called.

Each of these approaches is described in the following sections.

Using Polling to Prevent Blocking

To poll the read or write stream, you need to see if the streams are ready. When writing to a write stream, this is done by calling the function `CFWriteStreamCanAcceptBytes`. If it returns `TRUE`, then you can use the `CFWriteStreamWrite` function because it will be able to write immediately and will not block. Similarly, for a read stream, before calling `CFReadStreamRead`, call the function `CFReadStreamHasBytesAvailable`. By polling the stream, you avoid blocking the thread by waiting for the stream to become ready.

Listing 2-6 is a polling example for a read stream.

Listing 2-6   Polling a read stream

```
while (!done) {
    if (CFReadStreamHasBytesAvailable(myReadStream)) {
        UInt8 buf[BUFSIZE];
        CFIndex bytesRead = CFReadStreamRead(myReadStream, buf, BUFSIZE);
        if (bytesRead < 0) {
            CFStreamError error = CFReadStreamGetError(myReadStream);
            reportError(error);
        } else if (bytesRead == 0) {
            if (CFReadStreamGetStatus(myReadStream) == kCFStreamStatusAtEnd) {
                done = TRUE;
            }
        } else {
            handleBytes(buf, bytesRead);
        }
    } else {
        // ...do something else while you wait...
    }
}
```

Listing 2-7 is a polling example for a write stream.

Listing 2-7   Polling a write stream

```
UInt8 buf[] = "Hello, world";
UInt32 bufLen = strlen(buf);

while (!done) {
    if (CFWriteStreamCanAcceptBytes(myWriteStream)) {
        int bytesWritten = CFWriteStreamWrite(myWriteStream, buf, strlen(buf));
        if (bytesWritten < 0) {
            CFStreamError error = CFWriteStreamGetError(myWriteStream);
            reportError(error);
        } else if (bytesWritten == 0) {
            if (CFWriteStreamGetStatus(myWriteStream) == kCFStreamStatusAtEnd)
            {
                done = TRUE;
            }
        } else if (bytesWritten != strlen(buf)) {
            // Determine how much has been written and adjust the buffer
            bufLen = bufLen - bytesWritten;
            memmove(buf, buf + bytesWritten, bufLen);

            // Figure out what went wrong with the write stream
            CFStreamError error = CFWriteStreamGetError(myWriteStream);
```

```
        reportError(error);
    }
} else {
    // ...do something else while you wait...
}
}
```

Using a Run Loop to Prevent Blocking

The run loop of a thread watches for certain events to happen. When those events take place the run loop calls a specified function. The run loop is constantly monitoring all of its input sources for events. In the case of network transfers, your callback function will be executed by the run loop when the event you registered for occurs. This allows you to not have to poll your socket stream, which would slow down the thread. Please make sure you are familiar with run loops by reading *Run Loops*.

This example begins by creating a socket read stream:

```
CFStreamCreatePairWithSocketToCFHost(kCFAllocatorDefault, host, port,
                                     &myReadStream, NULL);
```

where the CFHost object reference, `host`, specifies the remote host with which the read stream is to be made and the `port` parameter specifies the port number that the host uses. The `CFStreamCreatePairWithSocketToCFHost` function returns the new read stream reference in `myReadStream`. The last parameter, `NULL`, indicates that the caller does not want to create a write stream. If you wanted to create a write steam, the last parameter would be, for example, `&myWriteStream`.

Before opening the socket read stream, create a context that will be used when you register to receive stream-related events:

```
CFStreamClientContext myContext = {0, myPtr, myRetain, myRelease, myCopyDesc};
```

The first parameter is `0` to specify the version number. The `info` parameter, `myPtr`, is a pointer to data you want to be passed to your callback function. Usually, `myPtr` is a pointer to a structure you've defined that contains information relating to the stream. The `retain` parameter is a pointer to a function to retain the `info` parameter. So if you set it to your function `myRetain`, as in the code above, CFStream will call `myRetain(myPtr)` to retain the `info` pointer. Similarly, the `release` parameter, `myRelease`, is a pointer to a function to release the info parameter. When the stream is disassociated from the context, CFStream would call `myRelease(myPtr)`. Finally, `copyDescription` is a parameter to a function to provide a description of the stream. For example, if you were to call `CFCopyDesc(myReadStream)` with the stream client context shown above, CFStream would call `myCopyDesc(myPtr)`.

The client context also allows you the option of setting the `retain`, `release`, and `copyDescription` parameters to `NULL`. If you set the `retain` and `release` parameters to `NULL`, then the system will expect you to keep the memory pointed to by the `info` pointer alive until the stream itself is destroyed. If you set the `copyDescription` parameter to `NULL`, then the system will provide, if requested, a rudimentary description of what is in the memory pointed to by the `info` pointer.

With the client context set up, call the function `CFReadStreamSetClient` to register to receive stream-related events. `CFReadStreamSetClient` requires that you specify the callback function and the events you want to receive. The following example in Listing 2-8 specifies that the callback function wants to receive the `kCFStreamEventHasBytesAvailable`, `kCFStreamEventErrorOccurred`, and `kCFStreamEventEndEncountered` events. Then schedule the stream on a run loop with the `CFReadStreamScheduleWithRunLoop` function. See Listing 2-8 for an example of how to do this.

Listing 2-8   Scheduling a stream on a run loop

```
CFOptionFlags registeredEvents = kCFStreamEventHasBytesAvailable |
        kCFStreamEventErrorOccurred | kCFStreamEventEndEncountered;
if (CFReadStreamSetClient(myReadStream, registeredEvents, myCallBack, &myContext)
{
    CFReadStreamScheduleWithRunLoop(myReadStream, CFRunLoopGetCurrent(),
                                    kCFRunLoopCommonModes);
}
```

With the stream scheduled on the run loop, you are ready to open the stream as shown in Listing 2-9.

Listing 2-9   Opening a nonblocking read stream

```
if (!CFReadStreamOpen(myReadStream)) {
    CFStreamError myErr = CFReadStreamGetError(myReadStream);
    if (myErr.error != 0) {
    // An error has occurred.
        if (myErr.domain == kCFStreamErrorDomainPOSIX) {
        // Interpret myErr.error as a UNIX errno.
            strerror(myErr.error);
        } else if (myErr.domain == kCFStreamErrorDomainMacOSStatus) {
            OSStatus macError = (OSStatus)myErr.error;
            }
        // Check other domains.
    } else
        // start the run loop
        CFRunLoopRun();
}
```

Now, wait for your callback function to be executed. In your callback function, check the event code and take appropriate action. See Listing 2-10.

Listing 2-10   Network events callback function

```
void myCallBack (CFReadStreamRef stream, CFStreamEventType event, void *myPtr)
 {
    switch(event) {
        case kCFStreamEventHasBytesAvailable:
            // It is safe to call CFReadStreamRead; it won't block because bytes
            // are available.
            UInt8 buf[BUFSIZE];
            CFIndex bytesRead = CFReadStreamRead(stream, buf, BUFSIZE);
            if (bytesRead > 0) {
                handleBytes(buf, bytesRead);
            }
            // It is safe to ignore a value of bytesRead that is less than or
            // equal to zero because these cases will generate other events.
            break;
        case kCFStreamEventErrorOccurred:
            CFStreamError error = CFReadStreamGetError(stream);
            reportError(error);
            CFReadStreamUnscheduleFromRunLoop(stream, CFRunLoopGetCurrent(),
                                              kCFRunLoopCommonModes);
            CFReadStreamClose(stream);
            CFRelease(stream);
            break;
```

```
        case kCFStreamEventEndEncountered:
            reportCompletion();
            CFReadStreamUnscheduleFromRunLoop(stream, CFRunLoopGetCurrent(),
                                              kCFRunLoopCommonModes);
            CFReadStreamClose(stream);
            CFRelease(stream);
            break;
    }
}
```

When the callback function receives the `kCFStreamEventHasBytesAvailable` event code, it calls `CFReadStreamRead` to read the data.

When the callback function receives the `kCFStreamEventErrorOccurred` event code, it calls `CFReadStreamGetError` to get the error and its own error function (`reportError`) to handle the error.

When the callback function receives the `kCFStreamEventEndEncountered` event code, it calls its own function (`reportCompletion`) for handling the end of data and then calls the `CFReadStreamUnscheduleFromRunLoop` function to remove the stream from the specified run loop. Then the `CFReadStreamClose` function is run to close the stream and `CFRelease` to release the stream reference.

Navigating Firewalls

There are two ways to apply firewall settings to a stream. For most streams, you can retrieve the proxy settings using the `SCDynamicStoreCopyProxies` function and then apply the result to the stream by setting the `kCFStreamHTTPProxy` (or `kCFStreamFTPProxy`) property. The `SCDynamicStoreCopyProxies` function is part of the System Configuration framework, so you need to include `<SystemConfiguration/SystemConfiguration.h>` in your project to use the function. Then just release the proxy dictionary reference when you are done with it. The process would look like that in Listing 2-11.

Listing 2-11   Navigating a stream through a proxy server

```
CFDictionaryRef proxyDict = SCDynamicStoreCopyProxies(NULL);
CFReadStreamSetProperty(readStream, kCFStreamPropertyHTTPProxy, proxyDict);
```

However, if you need to use the proxy settings often for multiple streams, it becomes a bit more complicated. In this case retrieving the firewall settings of a user's machine requires five steps:

1. Create a single, persistent handle to a dynamic store session, `SCDynamicStoreRef`.

2. Put the handle to the dynamic store session into the run loop to be notified of proxy changes.

3. Use `SCDynamicStoreCopyProxies` to retrieve the latest proxy settings.

4. Update your copy of the proxies when told of the changes.

5. Clean up the `SCDynamicStoreRef` when you are through with it.

To create the handle to the dynamic store session, use the function `SCDynamicStoreCreate` and pass an allocator, a name to describe your process, a callback function and a dynamic store context, `SCDynamicStoreContext`. This is run when initializing your application. The code would be similar to that in Listing 2-12.

Listing 2-12   Creating a handle to a dynamic store session

```
SCDynamicStoreContext context = {0, self, NULL, NULL, NULL};
systemDynamicStore = SCDynamicStoreCreate(NULL,
                                          CFSTR("SampleApp"),
                                          proxyHasChanged,
                                          &context);
```

After creating the reference to the dynamic store, you need to add it to the run loop. First, take the dynamic store reference and set it up to monitor for any changes to the proxies. This is accomplished with the functions `SCDynamicStoreKeyCreateProxies` and `SCDynamicStoreSetNotificationKeys`. Then, you can add the dynamic store reference to the run loop with the functions `SCDynamicStoreCreateRunLoopSource` and `CFRunLoopAddSource`. Your code should look like that in Listing 2-13.

Listing 2-13   Adding a dynamic store reference to the run loop

```
// Set up the store to monitor any changes to the proxies
CFStringRef proxiesKey = SCDynamicStoreKeyCreateProxies(NULL);
CFArrayRef keyArray = CFArrayCreate(NULL,
                                    (const void **)(&proxiesKey),
                                    1,
                                    &kCFTypeArrayCallBacks);
SCDynamicStoreSetNotificationKeys(systemDynamicStore, keyArray, NULL);
CFRelease(keyArray);
CFRelease(proxiesKey);

// Add the dynamic store to the run loop
CFRunLoopSourceRef storeRLSource =
    SCDynamicStoreCreateRunLoopSource(NULL, systemDynamicStore, 0);
CFRunLoopAddSource(CFRunLoopGetCurrent(), storeRLSource, kCFRunLoopCommonModes);
CFRelease(storeRLSource);
```

Once the dynamic store reference has been added to the run loop, use it to preload the proxy dictionary the current proxy settings by calling `SCDynamicStoreCopyProxies`. See Listing 2-14 for how to do this.

Listing 2-14   Loading the proxy dictionary

```
gProxyDict = SCDynamicStoreCopyProxies(systemDynamicStore);
```

As a result of adding the dynamic store reference to the run loop, each time the proxies are changed your callback function will be run. Release the current proxy dictionary and reload it with the new proxy settings. A sample callback function would look like the one in Listing 2-15.

Listing 2-15   Proxy callback function

```
void proxyHasChanged() {
    CFRelease(gProxyDict);
    gProxyDict = SCDynamicStoreCopyProxies(systemDynamicStore);
}
```

Since all of the proxy information is up-to-date, apply the proxies. After creating your read or write stream, set the `kCFStreamPropertyHTTPProxy` proxy by calling the functions `CFReadStreamSetProperty` or `CFWriteStreamSetProperty`. If your stream was a read stream called `readStream`, your function call would be like that in Listing 2-16.

Listing 2-16   Adding proxy information to a stream

```
CFReadStreamSetProperty(readStream, kCFStreamPropertyHTTPProxy, gProxyDict);
```

When you are all done with using the proxy settings, make sure to release the dictionary and dynamic store reference, and to remove the dynamic store reference from the run loop. See Listing 2-17.

Listing 2-17   Cleaning up proxy information

```
if (gProxyDict) {
    CFRelease(gProxyDict);
}

// Invalidate the dynamic store's run loop source
// to get the store out of the run loop
CFRunLoopSourceRef rls = SCDynamicStoreCreateRunLoopSource(NULL,
systemDynamicStore, 0);
CFRunLoopSourceInvalidate(rls);
CFRelease(rls);
CFRelease(systemDynmaicStore);
```

Communicating with HTTP Servers

This chapter explains how to create, send, and receive HTTP requests and responses.

Creating a CFHTTP Request

An HTTP request is a message consisting of a method for the remote server to execute, the object to operate on (the URL), message headers, and a message body. The methods are usually one of the following: `GET`, `HEAD`, `PUT`, `POST`, `DELETE`, `TRACE`, `CONNECT` or `OPTIONS`. Creating an HTTP request with CFHTTP requires four steps:

1. Generate a CFHTTP message object using the `CFHTTPMessageCreateRequest` function.

2. Set the body of the message using the function `CFHTTPMessageSetBody`.

3. Set the message's headers using the `CFHTTPMessageSetHeaderFieldValue` function.

4. Serialize the message by calling the function `CFHTTPMessageCopySerializedMessage`.

Sample code would look like the code in Listing 3-1.

Listing 3-1   Creating an HTTP request

```
CFStringRef bodyData = CFSTR(""); // Usually used for POST data

CFStringRef headerFieldName = CFSTR("X-My-Favorite-Field");
CFStringRef headerFieldValue = CFSTR("Dreams");

CFStringRef url = CFSTR("http://www.apple.com");
CFURLRef myURL = CFURLCreateWithString(kCFAllocatorDefault, url, NULL);

CFStringRef requestMethod = CFSTR("GET");
CFHTTPMessageRef myRequest =
    CFHTTPMessageCreateRequest(kCFAllocatorDefault, requestMethod, myURL,
                               kCFHTTPVersion1_1);

CFHTTPMessageSetBody(myRequest, bodyData);
CFHTTPMessageSetHeaderFieldValue(myRequest, headerFieldName, headerFieldValue);
CFDataRef mySerializedRequest = CFHTTPMessageCopySerializedMessage(myRequest);
```

In this sample code, `url` is first converted into a CFURL object by calling `CFURLCreateWithString`. Then `CFHTTPMessageCreateRequest` is called with four parameters: `kCFAllocatorDefault` specifies that the default system memory allocator is to be used to create the message reference, `requestMethod` specifies the method, such as the POST method, `myURL` specifies the URL, such as `http://www.apple.com`, and `kCFHTTPVersion1_1` specifies that message's HTTP version is to be 1.1.

The message object reference (`myRequest`) returned by `CFHTTPMessageCreateRequest` is then sent to `CFHTTPMessageSetBody` along with the body of the message (`bodyData`). Then call `CFHTTPMessageSetHeaderFieldValue` using the same message object reference along with the name of the header (`headerField`), and the value to be set (`value`). The header parameter is a CFString object such as `Content-Length`, and the value parameter is a CFString object such as `1260`. Finally, the message is serialized by calling `CFHTTPMessageCopySerializedMessage` and should be sent via a write stream to the intended recipient, in this example `http://www.apple.com`.

> **Note:** The request body is usually omitted. The main place a request body is used is in a POST request to contain the POST data. It may also be used in some other request types related to HTTP extensions such as WebDAV. See RFC 2616 for more information.

When the message is no longer needed, release the message object and the serialized message. See Listing 3-2 for sample code.

Listing 3-2   Releasing an HTTP request

```
CFRelease(myRequest);
CFRelease(myURL);
CFRelease(url);
CFRelease(mySerializedRequest);
myRequest = NULL;
mySerializedRequest = NULL;
```

Creating a CFHTTP Response

The steps for creating an HTTP response are almost identical to those for creating an HTTP request. The only difference is that rather than calling `CFHTTPMessageCreateRequest`, you call the function `CFHTTPMessageCreateResponse` using the same parameters.

Deserializing an Incoming HTTP Request

To deserialize an incoming HTTP request, create an empty message using the `CFHTTPMessageCreateEmpty` function, passing `TRUE` as the `isRequest` parameter to specify that an empty request message is to be created. Then append the incoming message to the empty message using the function `CFHTTPMessageAppendBytes`. `CFHTTPMessageAppendBytes` deserializes the message and removes any control information it may contain. Continue to do this until the function `CFHTTPMessageIsHeaderComplete` returns `TRUE`. If you do not check for `CFHTTPMessageIsHeaderComplete` to return `TRUE`, the message may be incomplete and unreliable. A sample of using these two functions can be seen in Listing 3-3.

Listing 3-3   Deserializing a message

```
CFHTTPMessageRef myMessage = CFHTTPMessageCreateEmpty(kCFAllocatorDefault, TRUE);
if (!CFHTTPMessageAppendBytes(myMessage, &data, numBytes)) {
    //Handle parsing error
}
```

In the example, `data` is the data that is to be appended and `numBytes` is the length of `data`. You may want to call `CFHTTPMessageIsHeaderComplete` to verify that the header of the appended message is complete.

```
if (CFHTTPMessageIsHeaderComplete(myMessage)) {
    // Perform processing.
}
```

With the message deserialized, you can now call any of the following functions to extract information from the message:

- `CFHTTPMessageCopyBody` to get a copy of the message's body

- `CFHTTPMessageCopyHeaderFieldValue` to get a copy of a specific header field value

- `CFHTTPMessageCopyAllHeaderFields` to get a copy of all of the message's header fields

- `CFHTTPMessageCopyRequestURL` to get a copy of the message's URL

- `CFHTTPMessageCopyRequestMethod` to get a copy of the message's request method

When you no longer need the message, release and dispose of it properly.

Deserializing an Incoming HTTP Response

Just as creating an HTTP request is very similar to creating an HTTP response, deserializing an incoming HTTP request is also very similar to deserializing an incoming HTTP response. The only important difference is that when calling `CFHTTPMessageCreateEmpty`, you must pass `FALSE` as the `isRequest` parameter to specify that the message to be created is a response message.

Using a Read Stream to Serialize and Send HTTP Requests

You can use a CFReadStream object to serialize and send CFHTTP requests. When you use a CFReadStream object to send a CFHTTP request, opening the stream causes the message to be serialized and sent in one step. Using a CFReadStream object to send CFHTTP requests makes it easy to get the response to the request because the response is available as a property of the stream.

Serializing and Sending an HTTP Request

To use a CFReadStream object to serialize and send an HTTP request, first create a CFHTTP request and set the message body and headers as described in . Then create a CFReadStream object by calling the function `CFReadStreamCreateForHTTPRequest` and passing the request you just created. Finally, open the read stream with `CFReadStreamOpen`.

When `CFReadStreamCreateForHTTPRequest` is called, it makes a copy of the CFHTTP request object that it is passed. Thus, if necessary, you could release the CFHTTP request object immediately after calling `CFReadStreamCreateForHTTPRequest`.

Because the read stream opens a socket connection with the server specified by the `myUrl` parameter when the CFHTTP request was created, some amount of time must be allowed to pass before the stream is considered to be open. Opening the read stream also causes the request to be serialized and sent.

A sample of how to serialize and send an HTTP request can be seen in Listing 3-4.

Listing 3-4   Serializing an HTTP request with a read stream

```
CFHTTPMessageRef myRequest = CFHTTPMessageCreateRequest(kCFAllocatorDefault,
        requestMethod, myUrl, kCFHTTPVersion1_1);
CFHTTPMessageSetBody(myRequest, bodyData);
CFHTTPMessageSetHeaderFieldValue(myRequest, headerField, value);

CFReadStreamRef myReadStream =
CFReadStreamCreateForHTTPRequest(kCFAllocatorDefault, myRequest);

CFReadStreamOpen(myReadStream);
```

Checking the Response

Call `CFReadStreamCopyProperty` to get the message response from the read stream:

```
CFHTTPMessageRef myResponse = CFReadStreamCopyProperty(myReadStream,
kCFStreamPropertyHTTPResponseHeader);
```

You can get the complete status line from the response message by calling the function `CFHTTPMessageCopyResponseStatusLine`:

```
CFStringRef myStatusLine = CFHTTPMessageCopyResponseStatusLine(myResponse);
```

Or get just the status code from the response message by calling the function `CFHTTPMessageGetResponseStatusCode`:

```
UInt32 myErrCode = CFHTTPMessageGetResponseStatusCode(myResponse);
```

Handling Authentication Errors

If the status code returned by the function `CFHTTPMessageGetResponseStatusCode` is 401 (the remote server requires authentication information) or 407 (a proxy server requires authentication), you need to append authentication information to the request and send it again. Please read "Communicating with Authenticating HTTP Servers" (page 33) for information on how to handle authentication.

Handling Redirection Errors

When `CFReadStreamCreateForHTTPRequest` creates a read stream, automatic redirection for the stream is disabled by default. If the uniform resource locator, or URL, to which the request is sent is redirected to another URL, sending the request will result in an error whose status code ranges from 300 to 307. If you receive a redirection error, you need to close the stream, create the stream again, enable automatic redirection for it, and open the stream. See Listing 3-5.

Listing 3-5   Redirecting an HTTP stream

```
CFReadStreamClose(myReadStream);
CFReadStreamRef myReadStream =
    CFReadStreamCreateForHTTPRequest(kCFAllocatorDefault, myRequest);
if (CFReadStreamSetProperty(myReadStream, kCFStreamPropertyHTTPShouldAutoredirect,
 kCFBooleanTrue) == false) {
    // something went wrong, exit
}
CFReadStreamOpen(myReadStream);
```

You may want to enable automatic redirection whenever you create a read stream.

Cancelling a Pending Request

Once a request has been sent, it is not possible to prevent the remote server from acting on it. However, if you no longer care about the response data, you can close the stream.

> **Important:** Do not close a stream from any thread while another thread is waiting for content from that stream. If you need to be able to terminate a request, you should use non-blocking I/O as described in "Preventing Blocking When Working with Streams" (page 19). Be sure to remove the stream from your run loop before closing it.

Cancelling a Pending Request

Communicating with Authenticating HTTP Servers

This chapter describes how to interact with authenticating HTTP servers by taking advantage of the CFHTTPAuthentication API. It explains how to find matching authentication objects and credentials, apply them to an HTTP request, and store them for later use.

In general, if an HTTP server returns a 401 or 407 response following your HTTP request, it means that the server is authenticating and requires credentials. In the CFHTTPAuthentication API, each set of credentials is stored in a CFHTTPAuthentication object. Therefore, every different authenticating server and every different user connecting to that server requires a separate CFHTTPAuthentication object. To communicate with the server, you need to apply your CFHTTPAuthentication object to the HTTP request. These steps are explained in more detail next.

Handling Authentication

Adding support for authentication will allow your application to talk with authenticating HTTP servers (if the server returns a 401 or 407 response). Even though HTTP authentication is not a difficult concept, it is a complicated process to execute. The procedure is as follows:

1. The client sends an HTTP request to the server.

2. The server returns a challenge to the client.

3. The client bundles the original request with credentials and sends them back to the server.

4. A negotiation takes place between the client and server.

5. When the server has authenticated the client, it sends back the response to the request.

Performing this procedure requires a number of steps. A diagram of the entire procedure can be seen in Figure 4-1 and Figure 4-2.

Figure 4-1   Handling authentication
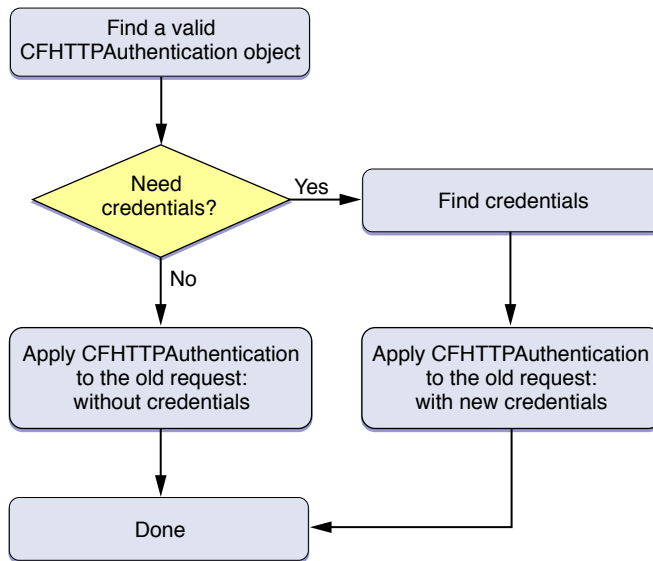
Figure 4-2   Finding an authentication object
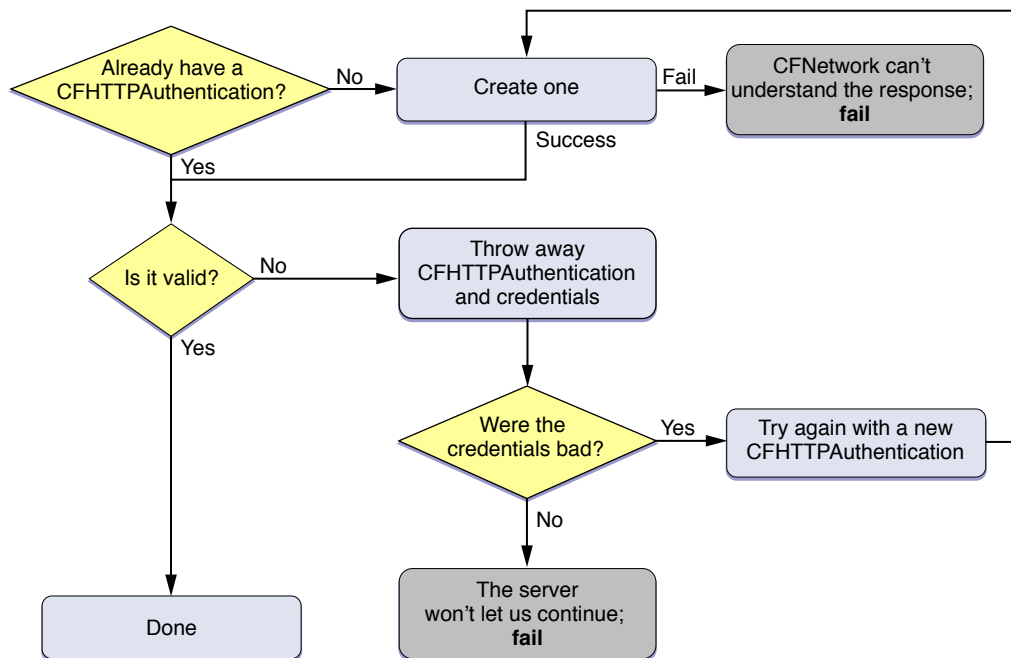
When an HTTP request returns a 401 or 407 response, the first step is for the client to find a valid CFHTTPAuthentication object. An authentication object contains credentials and other information that, when applied to an HTTP message request, verifies your identity with the server. If you've already authenticated once with the server, you will have a valid authentication object. However, in most cases, you will need to create this object from the response with the `CFHTTPAuthenticationCreateFromResponse` function. See Listing 4-1.

> **Note:** All the sample code regarding authentication is adapted from the *ImageClient* application.

Listing 4-1   Creating an authentication object

```
if (!authentication) {
    CFHTTPMessageRef responseHeader =
        (CFHTTPMessageRef) CFReadStreamCopyProperty(
            readStream,
            kCFStreamPropertyHTTPResponseHeader
        );

    // Get the authentication information from the response.
    authentication = CFHTTPAuthenticationCreateFromResponse(NULL, responseHeader);
    CFRelease(responseHeader);
}
```

If the new authentication object is valid, then you are done and can continue to the second step of Figure 4-1 (page 34). If the authentication object is not valid, then throw away the authentication object and credentials and check to see if the credentials were bad. For more information about credentials, read "Security Credentials".

Bad credentials mean that the server did not accept the login information at it will continue to listen for new credentials. However, if the credentials were good but the server still rejected your request, then the server is refusing to speak with you, so you must give up. Assuming the credentials were bad, retry this entire process beginning with creating an authentication object until you get working credentials and a valid authentication object. In code, this procedure should look like the one in Listing 4-2.

Listing 4-2   Finding a valid authentication object

```
CFStreamError err;
if (!authentication) {
    // the newly created authentication object is bad, must return
    return;

} else if (!CFHTTPAuthenticationIsValid(authentication, &err)) {

    // destroy authentication and credentials
    if (credentials) {
        CFRelease(credentials);
        credentials = NULL;
    }
    CFRelease(authentication);
    authentication = NULL;

    // check for bad credentials (to be treated separately)
    if (err.domain == kCFStreamErrorDomainHTTP &&
        (err.error == kCFStreamErrorHTTPAuthenticationBadUserName
        || err.error == kCFStreamErrorHTTPAuthenticationBadPassword))
    {
        retryAuthorizationFailure(&authentication);
        return;
    } else {
        errorOccurredLoadingImage(err);
    }
}
```

Now that you have a valid authentication object, continue following the flowchart in Figure 4-1 (page 34). First, determine whether you need credentials. If you don't, then apply the authentication object to the HTTP request. The authentication object is applied to the HTTP request in Listing 4-4 (page 37) (`resumeWithCredentials`).

Without storing credentials (as explained in "Keeping Credentials in Memory" (page 37) and "Keeping Credentials in a Persistent Store" (page 38)), the only way to obtain valid credentials is by prompting the user. Most of the time, a user name and password are needed for the credentials. By passing the authentication object to the `CFHTTPAuthenticationRequiresUserNameAndPassword` function you can see if a user name and password are necessary. If the credentials do need a user name and password, prompt the user for them and store them in the credentials dictionary. For an NTLM server, the credentials also require a domain. After you have the new credentials, you can apply the authentication object to the HTTP request using the `resumeWithCredentials` function from Listing 4-4 (page 37). This whole process is shown in Listing 4-3.

> **Note:** In code listings, when comments are preceded and succeeded by ellipses, it means that that action is outside the scope of this document, but does need to be implemented. This is different from normal comments which describe what action is taking place.

Listing 4-3   Finding credentials (if necessary) and applying them

```
// ...continued from Listing 4-2
else {
    cancelLoad();
    if (credentials) {
        resumeWithCredentials();
    }
    // are a user name & password needed?
    else if (CFHTTPAuthenticationRequiresUserNameAndPassword(authentication))
        {
        CFStringRef realm = NULL;
        CFURLRef url = CFHTTPMessageCopyRequestURL(request);

        // check if you need an account domain so you can display it if necessary
        if (!CFHTTPAuthenticationRequiresAccountDomain(authentication)) {
            realm = CFHTTPAuthenticationCopyRealm(authentication);
        }
        // ...prompt user for user name (user), password (pass)
        // and if necessary domain (domain) to give to the server...

        // Guarantee values
        if (!user) user = (CFStringRef)@"";
        if (!pass) pass = (CFStringRef)@"";

        CFDictionarySetValue(credentials, kCFHTTPAuthenticationUsername, user);
        CFDictionarySetValue(credentials, kCFHTTPAuthenticationPassword, pass);

        // Is an account domain needed? (used currently for NTLM only)
        if (CFHTTPAuthenticationRequiresAccountDomain(authentication)) {
            if (!domain) domain = (CFStringRef)@"";
            CFDictionarySetValue(credentials,
                            kCFHTTPAuthenticationAccountDomain, domain);
        }
        if (realm) CFRelease(realm);
        CFRelease(url);
```

```
    }
    else {
        resumeWithCredentials();
    }
}
```

Listing 4-4   Applying the authentication object to a request

```
void resumeWithCredentials() {
    // Apply whatever credentials we've built up to the old request
    if (!CFHTTPMessageApplyCredentialDictionary(request, authentication,
                                               credentials, NULL)) {
        errorOccurredLoadingImage();
    } else {
        // Now that we've updated our request, retry the load
        loadRequest();
    }
}
```

Keeping Credentials in Memory

If you plan on communicating with an authenticating server often, it may be worth reusing credentials to avoid prompting the user for the server's user name and password multiple times. This section explains the changes that should be made to one-time use authentication code (such as in "Handling Authentication" (page 33)) to store credentials in memory for reuse later.

To reuse credentials, there are three data structure changes you need to make to your code.

1. Create a mutable array to hold all the authentication objects.

    ```
    CFMutableArrayRef authArray;
    ```

    instead of:

    ```
    CFHTTPAuthenticationRef authentication;
    ```

2. Create a mapping from authentication objects to credentials using a dictionary.

    ```
    CFMutableDictionaryRef credentialsDict;
    ```

    instead of:

    ```
    CFMutableDictionaryRef credentials;
    ```

3. Maintain these structures everywhere you used to modify the current authentication object and the current credentials.

    ```
    CFDictionaryRemoveValue(credentialsDict, authentication);
    ```

    instead of:

    ```
    CFRelease(credentials);
    ```

Now, after creating the HTTP request, look for a matching authentication object before each load. A simple, unoptimized method for finding the appropriate object can be seen in Listing 4-5.

Listing 4-5   Looking for a matching authentication object

```
CFHTTPAuthenticationRef findAuthenticationForRequest {
    int i, c = CFArrayGetCount(authArray);
    for (i = 0; i < c; i ++) {
        CFHTTPAuthenticationRef auth = (CFHTTPAuthenticationRef)
                CFArrayGetValueAtIndex(authArray, i);
        if (CFHTTPAuthenticationAppliesToRequest(auth, request)) {
            return auth;
        }
    }
    return NULL;
}
```

If the authentication array has a matching authentication object, then check the credentials store to see if the correct credentials are also available. Doing so prevents you from having to prompt the user for a user name and password again. Look for the credentials using the `CFDictionaryGetValue` function as shown in Listing 4-6.

Listing 4-6   Searching the credentials store

```
credentials = CFDictionaryGetValue(credentialsDict, authentication);
```

Then apply your matching authentication object and credentials to your original HTTP request and resend it.

⚠️ **Warning:** Do not apply credentials to the HTTP request *before* receiving a server challenge. The server may have changed since the last time you authenticated and you could create a security risk.

With these changes, you application will be able to store authentication objects and credentials in memory for use later.

Keeping Credentials in a Persistent Store

Storing credentials in memory prevents a user from having to reenter a server's user name and password during that specific application launch. However, when the application quits, those credentials will be released. To avoid losing the credentials, save them in a persistent store so each server's credentials need to be generated only once. A keychain is the recommended place for storing credentials. Even though you can have multiple keychains, this document refers to the user's default keychain as *the* keychain. Using the keychain means that the authentication information that you store can also be used in other applications trying to access the same server, and vice versa.

Storing and retrieving credentials in the keychain requires two functions: one for finding the credentials dictionary for authentication and one for saving the credentials of the most recent request. These functions will be declared in this document as:

```
CFMutableDictionaryRef findCredentialsForAuthentication(
        CFHTTPAuthenticationRef auth);
```

```
void saveCredentialsForRequest(void);
```

The function `findCredentialsForAuthentication` first checks the credentials dictionary stored in memory to see whether the credentials are cached locally. See Listing 4-6 (page 38) for how to implement this.

If the credentials are not cached in memory, then search the keychain. To search the keychain, use the function `SecKeychainFindInternetPassword`. This function requires a large number of parameters. The parameters, and a short description of how they are used with HTTP authentication credentials, are:

keychainOrArray
> `NULL` to specify the user's default keychain list.

serverNameLength
> The length of `serverName`, usually `strlen(serverName)`.

serverName
> The server name parsed from the HTTP request.

securityDomainLength
> The length of security domain, or `0` if there is no domain. In the sample code, `realm ? strlen(realm) : 0` is passed to account for both situations.

securityDomain
> The realm of the authentication object, obtained from the `CFHTTPAuthenticationCopyRealm` function.

accountNameLength
> The length of `accountName`. Since the `accountName` is `NULL`, this value is `0`.

accountName
> There is no account name when fetching the keychain entry, so this should be `NULL`.

pathLength
> The length of `path`, or `0` if there is no path. In the sample code, `path ? strlen(path) : 0` is passed to account for both situations.

path
> The path from the authentication object, obtained from the `CFURLCopyPath` function.

port
> The port number, obtained from the function `CFURLGetPortNumber`.

protocol
> A string representing the protocol type, such as HTTP or HTTPS. The protocol type is obtained by calling the `CFURLCopyScheme` function.

authenticationType
> The authentication type, obtained from the function `CFHTTPAuthenticationCopyMethod`.

passwordLength
> `0`, because no password is necessary when fetching a keychain entry.

passwordData
> `NULL`, because no password is necessary when fetching a keychain entry.

itemRef
> The keychain item reference object, `SecKeychainItemRef`, returned upon finding the correct keychain entry

When called properly, the code should look like that in Listing 4-7.

Listing 4-7   Searching the keychain

```
didFind =
    SecKeychainFindInternetPassword(NULL,
                                    strlen(host), host,
                                    realm ? strlen(realm) : 0, realm,
                                    0, NULL,
                                    path ? strlen(path) : 0, path,
                                    port,
                                    protocolType,
                                    authenticationType,
                                    0, NULL,
                                    &itemRef);
```

Assuming that `SecKeychainFindInternetPassword` returns successfully, create a keychain attribute list
(`SecKeychainAttributeList`) containing a single keychain attribute (`SecKeychainAttribute`). The
keychain attribute list will contain the user name and password. To load the keychain attribute list, call the
function `SecKeychainItemCopyContent` and pass it the keychain item reference object (`itemRef`) that
was returned by `SecKeychainFindInternetPassword`. This function will fill the keychain attribute with
the account's user name, and a `void **` as its password.

The user name and password can then be used to create a new set of credentials. Listing 4-8 shows this
procedure.

Listing 4-8   Loading server credentials from the keychain

```
if (didFind == noErr) {

    SecKeychainAttribute      attr;
    SecKeychainAttributeList attrList;
    UInt32                    length;
    void                      *outData;

    // To set the account name attribute
    attr.tag = kSecAccountItemAttr;
    attr.length = 0;
    attr.data = NULL;

    attrList.count = 1;
    attrList.attr = &attr;

    if (SecKeychainItemCopyContent(itemRef, NULL, &attrList, &length, &outData)
        == noErr) {

        // attr.data is the account (username) and outdata is the password
        CFStringRef username =
            CFStringCreateWithBytes(kCFAllocatorDefault, attr.data,
                                    attr.length, kCFStringEncodingUTF8, false);
        CFStringRef password =
            CFStringCreateWithBytes(kCFAllocatorDefault, outData, length,
                                    kCFStringEncodingUTF8, false);
        SecKeychainItemFreeContent(&attrList, outData);

        // create credentials dictionary and fill it with the user name & password
        credentials =
            CFDictionaryCreateMutable(NULL, 0,
                                      &kCFTypeDictionaryKeyCallBacks,
                                      &kCFTypeDictionaryValueCallBacks);
```

```
        CFDictionarySetValue(credentials, kCFHTTPAuthenticationUsername,
                             username);
        CFDictionarySetValue(credentials, kCFHTTPAuthenticationPassword,
                             password);

        CFRelease(username);
        CFRelease(password);
    }
    CFRelease(itemRef);
}
```

Retrieving credentials from the keychain is only useful if you can store credentials in the keychain first. The steps are very similar to loading credentials. First, see if the credentials are already stored in the keychain. Call `SecKeychainFindInternetPassword`, but pass the user name for `accountName` and the length of `accountName` for `accountNameLength`.

If the entry exists, modify it to change the password. Set the `data` field of the keychain attribute to contain the user name, so that you modify the correct attribute. Then call the function `SecKeychainItemModifyContent` and pass the keychain item reference object (`itemRef`), the keychain attribute list, and the new password. By modifying the keychain entry rather than overwriting it, the keychain entry will be properly updated and any associated metadata will still be preserved. The entry should look like the one in Listing 4-9.

Listing 4-9   Modifying the keychain entry

```
// Set the attribute to the account name
attr.tag = kSecAccountItemAttr;
attr.length = strlen(username);
attr.data = (void*)username;

// Modify the keychain entry
SecKeychainItemModifyContent(itemRef, &attrList, strlen(password),
                             (void *)password);
```

If the entry does not exist, then you will need to create it from scratch. The function `SecKeychainAddInternetPassword` accomplishes this task. Its parameters are the same as `SecKeychainFindInternetPassword`, but in contrast with the call to `SecKeychainFindInternetPassword`, you supply `SecKeychainAddInternetPassword` both a user name and a password. Release the keychain item reference object following a successful call to `SecKeychainAddInternetPassword` unless you need to use it for something else. See the function call in Listing 4-10.

Listing 4-10   Storing a new keychain entry

```
SecKeychainAddInternetPassword(NULL,
                               strlen(host), host,
                               realm ? strlen(realm) : 0, realm,
                               strlen(username), username,
                               path ? strlen(path) : 0, path,
                               port,
                               protocolType,
                               authenticationType,
                               strlen(password), password,
                               &itemRef);
```

Authenticating Firewalls

Authenticating firewalls is very similar to authenticating servers except that every failed HTTP request must be checked for both proxy authentication and server authentication. This means that you need separate stores (both local and persistent) for proxy servers and origin servers. Thus, the procedure for a failed HTTP response will now be:

■   Determine whether the response's status code was 407 (a proxy challenge). If it is, find a matching authentication object and credentials by checking the local proxy store and the persistent proxy store. If neither of those has a matching object and credentials, then request the credentials from the user. Apply the authentication object to the HTTP request and try again.

■   Determine whether the response's status code was 401 (a server challenge). If it is, follow the same procedure as with a 407 response, but use the origin server stores.

There are also a few minor differences to enforce when using proxy servers. The first is that the arguments to the keychain calls come from the proxy host and port, rather than from the URL for an origin server. The second is that when asking the user for a user name and password, make sure the prompt clearly states what the password is for.

By following these instructions, your application should be able to work with authenticating firewalls.

Working with FTP Servers

This chapter explains how to use some of the basic features of the CFFTP API. Managing the FTP transactions is performed asynchronously, while managing the file transfer is implemented synchronously.

Downloading a File

Using CFFTP is very similar to using CFHTTP because they are both based on CFStream. As with any other API that uses CFStream asynchronously, downloading a file with CFFTP requires that you create a read stream for the file, and a callback function for that read stream. When the read stream receives data, the callback function will be run and you will need to appropriately download the bytes. This procedure should normally be performed using two functions: one to set up the streams and one to act as the callback function.

Setting Up the FTP Streams

Begin by creating a read stream using the `CFReadStreamCreateWithFTPURL` function and passing it the URL string of the file to be downloaded on the remote server. An example of a URL string might be `ftp://ftp.example.com/file.txt`. Note that the string contains the server name, the path, and the file. Next, create a write stream for the local location where the file will be downloaded. This is accomplished using the `CFWriteStreamCreateWithFile` function, passing the path where the file will be downloaded.

Since the write stream and the read stream need to stay in sync, it is a good idea to create a structure that contains all of the common information, such as the proxy dictionary, the file size, the number of bytes written, the number of bytes left over, and a buffer. This structure might look like that in Listing 5-1.

Listing 5-1   A stream structure

```
typedef struct MyStreamInfo {

    CFWriteStreamRef  writeStream;
    CFReadStreamRef   readStream;
    CFDictionaryRef   proxyDict;
    SInt64            fileSize;
    UInt32            totalBytesWritten;
    UInt32            leftOverByteCount;
    UInt8             buffer[kMyBufferSize];

} MyStreamInfo;
```

Initialize your structure with the read stream and write stream you just created. You can then define the `info` field of your stream client context (`CFStreamClientContext`) to point to your structure. This will become useful later.

Open your write stream with the `CFWriteStreamOpen` function so you can begin writing to the local file. To make sure the stream opens properly, call the function `CFWriteStreamGetStatus` and check whether it returns either `kCFStreamStatusOpen` or `kCFStreamStatusOpening`.

With the write stream open, associate a callback function with the read stream. Call the function `CFReadStreamSetClient` and pass the read stream, the network events your callback function should receive, the callback function's name and the `CFStreamClientContext` object. By having earlier set the `info` field of the stream client context, your structure will now be sent to your callback function whenever it is run.

Some FTP servers may require a user name, and some may also require a password. If the server you are accessing needs a user name for authentication, call the `CFReadStreamSetProperty` function and pass the read stream, `kCFStreamPropertyFTPUserName` for the property, and a reference to a `CFString` object containing the user name. In addition, if you need to set a password, set the `kCFStreamPropertyFTPPassword` property.

Some computers may also use FTP proxies. Retrieve the proxy settings in a dictionary by calling the `SCDynamicStoreCopyProxies` function and passing it `NULL`. The function then returns a dynamic store reference. Then set the `kCFStreamPropertyFTPProxy` property of the read stream, and pass the proxy dictionary as the value. This sets the proxy server, specifies the port, and returns a Boolean value indicating whether passive mode is enforced for the FTP stream.

In addition to the properties mentioned, there are a number of other properties available for FTP streams. The complete list follows.

- `kCFStreamPropertyFTPUserName` — user name to use to log in (settable and retrievable; do not set for anonymous FTP connections)

- `kCFStreamPropertyFTPPassword` — password to use to log in (settable and retrievable; do not set for anonymous FTP connections)

- `kCFStreamPropertyFTPUsePassiveMode` — whether to use passive mode (settable and retrievable)

- `kCFStreamPropertyFTPResourceSize` — the expected size of an item that is being downloaded, if available (retrievable; available only for FTP read streams)

- `kCFStreamPropertyFTPFetchResourceInfo` — whether to require that resource information, such as size, be required before starting a download (settable and retrievable); setting this property may impact performance

- `kCFStreamPropertyFTPFileTransferOffset` — file offset at which to start a transfer (settable and retrievable)

- `kCFStreamPropertyFTPAttemptPersistentConnection` — whether to try to reuse connections (settable and retrievable)

- `kCFStreamPropertyFTPProxy` — CFDictionary type that holds key-value pairs of proxy dictionary (settable and retrievable)

- `kCFStreamPropertyFTPProxyHost` — name of an FTP proxy host (settable and retrievable)

- `kCFStreamPropertyFTPProxyPort` — port number of an FTP proxy host (settable and retrievable)

After the correct properties have been assigned to the read stream, open the stream using the `CFReadStreamOpen` function. Assuming that this does not return an error, all the streams have been properly set up.

Implementing the Callback Function

Your callback function will receive three parameters: the read stream, the type of event, and your `MyStreamInfo` structure. The type of event determines what action must be taken.

The most common event is `kCFStreamEventHasBytesAvailable`, which is sent when the read stream has received bytes from the server. First, check how many bytes have been read by calling the `CFReadStreamRead` function. Make sure the return value is not less than zero (an error), or equal to zero (download has completed). If the return value is positive, then you can begin writing the data in the read stream to disk via the write stream.

Call the `CFWriteStreamWrite` function to write the data to the write stream. Sometimes `CFWriteStreamWrite` can return without writing all of the data from the read stream. For this reason, set up a loop to run as long as there is still data to be written. The code for this loop is in Listing 5-2, where `info` is the `MyStreamInfo` structure from "Setting up the Streams" (page 43). This method of writing to the write stream uses blocking streams. You can achieve better performance by making the write stream event driven, but the code is more complex.

Listing 5-2   Writing data to a write stream from the read stream

```
bytesRead = CFReadStreamRead(info->readStream, info->buffer, kMyBufferSize);

//...make sure bytesRead > 0 ...

bytesWritten = 0;
while (bytesWritten < bytesRead) {
    CFIndex result;

    result = CFWriteStreamWrite(info->writeStream, info->buffer + bytesWritten,
 bytesRead - bytesWritten);
    if (result <= 0) {
        fprintf(stderr, "CFWriteStreamWrite returned %ld\n", result);
        goto exit;
    }
    bytesWritten += result;
}
info->totalBytesWritten += bytesWritten;
```

Repeat this entire procedure as long as there are available bytes in the read stream.

The other two events you need to watch out for are `kCFStreamEventErrorOccurred` and `kCFStreamEventEndEncountered`. If an error occurs, retrieve the error using `CFReadStreamGetError` and then exit. If the end of the file occurs, then your download has completed and you can exit.

Make sure to remove all your streams after everything is completed and no other process is using the streams. First, close the write stream and set the client to `NULL`. Then unschedule the stream from the run loop and release it. Remove the streams from the run loop when you are done.

Uploading a File

Uploading a file is similar to downloading a file. As with downloading a file, you need a read stream and a write stream. However, when uploading a file, the read stream will be for the local file and the write stream will be for the remote file. Follow the instructions in "Setting up the Streams" (page 43), but wherever it refers to the read stream, adapt the code for a write stream and visa versa.

In the callback function, rather than looking for the `kCFStreamEventHasBytesAvailable` event, now look for the event `kCFStreamEventCanAcceptBytes`. First, read bytes from the file using the read stream and place the data into the buffer in `MyStreamInfo`. Then, run the `CFWriteStreamWrite` function to push bytes from the buffer into the write stream. `CFWriteStreamWrite` returns the number of bytes that have been written to the stream. If the number of bytes written to the stream is fewer than the number read from the file, calculate the leftover bytes and store them back into the buffer. During the next write cycle, if there are leftover bytes, write them to the write stream rather than loading new data from the read stream. Repeat this whole procedure as long as the write stream can accept bytes (`CFWriteStreamCanAcceptBytes`). See this loop in code in Listing 5-3.

Listing 5-3   Writing data to the write stream

```
do {
    // Check for leftover data
    if (info->leftOverByteCount > 0) {
        bytesRead = info->leftOverByteCount;
    } else {
        // Make sure there is no error reading from the file
        bytesRead = CFReadStreamRead(info->readStream, info->buffer,
                                kMyBufferSize);
        if (bytesRead < 0) {
            fprintf(stderr, "CFReadStreamRead returned %ld\n", bytesRead);
            goto exit;
        }
        totalBytesRead += bytesRead;
    }

    // Write the data to the write stream
     bytesWritten = CFWriteStreamWrite(info->writeStream, info->buffer,
bytesRead);
    if (bytesWritten > 0) {

        info->totalBytesWritten += bytesWritten;

        // Store leftover data until kCFStreamEventCanAcceptBytes event occurs
 again
        if (bytesWritten < bytesRead) {
            info->leftOverByteCount = bytesRead - bytesWritten;
            memmove(info->buffer, info->buffer + bytesWritten,
                    info->leftOverByteCount);
        } else {
            info->leftOverByteCount = 0;
        }
    } else {
        if (bytesWritten < 0)
            fprintf(stderr, "CFWriteStreamWrite returned %ld\n", bytesWritten);
        break;
    }
```

```
        } while (CFWriteStreamCanAcceptBytes(info->writeStream));
```

Also account for the `kCFStreamEventErrorOccurred` and `kCFStreamEventEndEncountered` events as you do when downloading a file.

Creating a Remote Directory

To create a directory on a remote server, set up a write stream as if you were going to be uploading a file. However, provide a directory path, not a file, for the `CFURL` object that is passed to the `CFWriteStreamCreateWithFTPURL` function. End the path with a forward slash. For example, a proper directory path would be `ftp://ftp.example.com/newDirectory/`, not `ftp://ftp.example.com/newDirectory/newFile.txt`. When the callback function is executed by the run loop, it sends the event `kCFStreamEventOpenCompleted`, which means the directory has been created.

Only one level of directories can be created with each call to `CFWriteStreamCreateWithFTPURL`. Also, a directory is created only if you have the correct permissions on the server.

Downloading a Directory Listing

Downloading a directory listing via FTP is slightly different from downloading or uploading a file. This is because the incoming data has to be parsed. First, set up a read stream to get the directory listing. This should be done as it was for downloading a file: create the stream, register a callback function, schedule the stream with the run loop (if necessary, set up user name, password and proxy information), and finally open the stream. In the following example you do not need both a read and a write stream when retrieving the directory listing, because the incoming data is going to the screen rather than a file.

In the callback function, watch for the `kCFStreamEventHasBytesAvailable` event. Prior to loading data from the read stream, make sure there is no leftover data in the stream from the previous time the callback function was run. Load the offset from the `leftOverByteCount` field of your `MyStreamInfo` structure. Then, read data from the stream, taking into account the offset you just calculated. The buffer size and number of bytes read should be calculated too. This is all accomplished in Listing 5-4.

Listing 5-4   Loading data for a directory listing

```
// If previous call had unloaded data
int offset = info->leftOverByteCount;

// Load data from the read stream, accounting for the offset
bytesRead = CFReadStreamRead(info->readStream, info->buffer + offset,
                            kMyBufferSize - offset);
if (bytesRead < 0) {
    fprintf(stderr, "CFReadStreamRead returned %ld\n", bytesRead);
    break;
} else if (bytesRead == 0) {
    break;
}
bufSize = bytesRead + offset;
totalBytesRead += bufSize;
```

After the data has been read to a buffer, set up a loop to parse the data. The data that is parsed is not necessarily the entire directory listing; it could (and probably will) be chunks of the listing. Create the loop to parse the data using the function `CFFTPCreateParsedResourceListing`, which should be passed the buffer of data, the size of the buffer, and a dictionary reference. It returns the number of bytes parsed. As long as this value is greater than zero, continue to loop. The dictionary that `CFFTPCreateParsedResourceListing` creates contains all the directory listing information; more information about the keys is available in "Setting up the Streams" (page 43).

It is possible for `CFFTPCreateParsedResourceListing` to return a positive value, but not create a parse dictionary. For example, if the end of the listing contains information that cannot be parsed, `CFFTPCreateParsedResourceListing` will return a positive value to tell the caller that data has been consumed. However, `CFFTPCreateParsedResourceListing` will not create a parse dictionary since it could not understand the data.

If a parse dictionary is created, recalculate the number of bytes read and the buffer size as shown in Listing 5-5.

Listing 5-5   Loading the directory listing and parsing it

```
do
{
    bufRemaining = info->buffer + totalBytesConsumed;

    bytesConsumed = CFFTPCreateParsedResourceListing(NULL, bufRemaining,
                                                     bufSize, &parsedDict);
    if (bytesConsumed > 0) {

        // Make sure CFFTPCreateParsedResourceListing was able to properly
        // parse the incoming data
        if (parsedDict != NULL) {
            // ...Print out data from parsedDict...
            CFRelease(parsedDict);
        }

        totalBytesConsumed += bytesConsumed;
        bufSize -= bytesConsumed;
        info->leftOverByteCount = bufSize;

    } else if (bytesConsumed == 0) {

        // This is just in case. It should never happen due to the large buffer
 size
        info->leftOverByteCount = bufSize;
        totalBytesRead -= info->leftOverByteCount;
        memmove(info->buffer, bufRemaining, info->leftOverByteCount);

    } else if (bytesConsumed == -1) {
        fprintf(stderr, "CFFTPCreateParsedResourceListing parse failure\n");
        // ...Break loop and cleanup...
    }

} while (bytesConsumed > 0);
```

When the stream has no more bytes available, clean up all the streams and remove them from the run loop.

Using Network Diagnostics

In many network-based applications, network-based errors may occur that are unrelated to your application. However, most users are probably unaware of why an application is failing. The CFNetDiagnostics API allows you a quick and easy way to help the user fix their network problems with little work on your end.

If your application is using a CFStream object, then create a network diagnostic reference (`CFNetDiagnosticRef`) by calling the function `CFNetDiagnosticCreateWithStreams`. `CFNetDiagnosticCreateWithStreams` takes an allocator, a read stream, and a write stream as arguments. If your application uses only a read stream or a write stream, the unused argument should be set to `NULL`.

You can also create a network diagnostic reference straight from a URL if no stream exists. To do this, call the `CFNetDiagnosticCreateWithURL` function and pass it an allocator, and the URL as a `CFURLRef`. It will return a network diagnostic reference for you to use.

To diagnose the problem through the Network Diagnostic Assistant, call the `CFNetDiagnosticDiagnoseProblemInteractively` function and pass the network diagnostic reference. Listing 6-1 shows how to use CFNetDiagnostics with streams implemented on a run loop.

Listing 6-1   Using the CFNetDiagnostics API when a stream error occurs

```
case kCFStreamEventErrorOccurred:
    CFNetDiagnosticRef diagRef =
        CFNetDiagnosticCreateWithStreams(NULL, stream, NULL);
    (void)CFNetDiagnosticDiagnoseProblemInteractively(diagRef);
    CFStreamError error = CFReadStreamGetError(stream);
    reportError(error);
    CFReadStreamClose(stream);
    CFRelease(stream);
    break;
```

CFNetworkDiagnostics also gives you the ability to retrieve the status of the problem rather than using the Network Diagnostic Assistant. This is accomplished by calling `CFNetDiagnosticCopyNetworkStatusPassively`, which returns a constant value such as `kCFNetDiagnosticConnectionUp` or `kCFNetDiagnosticConnectionIndeterminate`.

Document Revision History

This table describes the changes to *CFNetwork Programming Guide*.

Date	Notes
2009-05-06	Corrected typos.
2009-01-06	Miscellaneous edits.
2008-10-15	Made minor corrections to code samples.
2008-03-11	Made minor typographical corrections and clarifications.
2007-01-08	Updated information regarding CFReadStreamUnscheduleFromRunLoop.
2006-05-23	Updated information regarding CFReadStreamCreateForHTTPRequest.
2006-04-04	Updated sample code for communicating with HTTP servers.
2006-03-08	Made minor editorial corrections throughout.
2006-02-07	Updated content substantially and moved reference information to the new document "CFNetwork Reference." Changed the title from "CFNetwork Services Programming Guide."
2005-08-11	Corrected description of port parameter in CFNetServiceCreate.
2005-04-29	Updated for Mac OS X v10.4. Changed "Rendezvous" to "Bonjour." Changed title from "CFNetwork Services."
2004-02-01	Added description of CFFTP and CFHost and clarified the protocols that CFNetwork Services currently supports. Corrected sample code in the sections "Working With Write Streams and "Using a Run Loop to Prevent Blocking." Corrected the description of the `clientContext` parameter for the `CFReadStreamClientCallback` and `CFWriteStreamClientCallback` callbacks.