Stream Programming Guide

Networking & Internet

2009-12-16

Contents

Figures and Listings

Introduction to Stream Programming Guide for Cocoa

A stream is a fundamental abstraction in programming: a sequence of bits transmitted serially from one point to another point. Cocoa provides three classes to represent streams and facilitate their use in your programs: NSStream, NSInputStream, and NSOutputStream. With the instances of these classes you can read data from, and write data to, files and application memory. You can also use these objects in socket-based connections to exchange data with remote hosts. You can also subclass the stream classes to obtain specialized stream behavior.

Organization of This Document

This document includes the following articles:

- "Cocoa Streams" (page 9) gives an overview of the Cocoa stream classes, describing architecture, capabilities, and general usage.
- "Reading From Input Streams" (page 11) explains how to create and prepare a (non-socket) input-stream object. It also describes how to handle stream events generated by all types of NSInputStream objects.
- "Writing To Output Streams" (page 15) explains how to create and prepare a (non-socket) output-stream object. It also describes how to handle stream events generated by all types of NSOutputStream objects.
- "Polling Versus Run-Loop Scheduling" (page 19) discusses the relative merits of the two techniques used to avoid blocking when reading and writing to streams. It also illustrates how to poll for stream data using the API of the stream classes.
- "Handling Stream Errors" (page 21) describes how to handle errors that occur in stream processing.
- "Setting Up Socket Streams" (page 23) explains how to set up stream objects used to communicate with remote hosts via sockets.

See Also

You may find the following external resources helpful if you are implementing socket-based network streams:

- OpenSSL — http://www.openssl.org/
- Apache SSL — http://www.apache-ssl.org/
- SOCKS — http://tools.ietf.org/html/rfc1928

Cocoa Streams

Streams provide an easy way for a program to exchange data with a variety of media in a device-independent way. A stream is a contiguous sequence of bits transmitted serially over a communications path. It is unidirectional and hence, from the perspective of a program, a stream can be an input (or read) stream or an output (or write) stream. Except for ones that are file-based, streams are non-seekable; once stream data has been provided or consumed, it cannot be retrieved again from the stream.

Cocoa includes three stream-related classes: `NSStream`, `NSInputStream`, and `NSOutputStream`. `NSStream` is an abstract class that defines the fundamental interface and properties for all stream objects. `NSInputStream` and `NSOutputStream` are subclasses of `NSStream` and implement default input-stream and output-stream behavior. You can create `NSOutputStream` instances for stream data located in memory or written to a file or C buffer; you can create `NSInputStream` instances for stream data read from an `NSData` object or a file. You can also have `NSInputStream` and `NSOutputStream` objects at the end points of a socket-based network connection and you can use stream objects without loading all of the stream data into memory at once. Figure 1 illustrates the types of input-stream and output-stream objects in terms of their sources or destinations.

Figure 1   Sources and destinations of stream objects
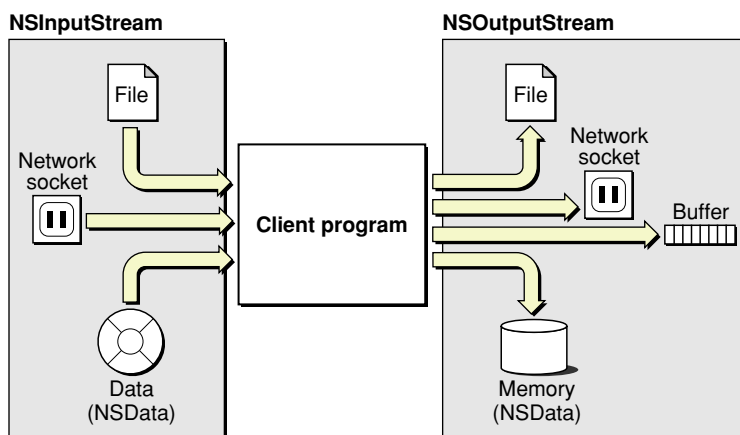
Because they deal with such a basic computing abstraction (streams), `NSStream` and its subclasses are intended for lower-level programming tasks. If there is a higher-level Cocoa API that is more suited for a particular task (for example, `NSURL` or `NSFileHandle`) use it instead.

Stream objects have properties associated with them. Most properties have to do with network security and configuration, namely secure-socket (SSL) levels and SOCKS proxy information. Two important additional properties are `NSStreamDataWrittenToMemoryStreamKey`, which permits retrieval of data written to memory for an output stream, and `NSStreamFileCurrentOffsetKey`, which allows you to manipulate the current read or write position in file-based streams.

A stream object also has a delegate associated with it. If a delegate is not explicitly set, the stream object itself becomes the delegate (a useful convention for custom subclasses). A stream object invokes the sole delegation method `stream:handleEvent:` for each stream-related event it handles. Of particular importance

are the events that indicate when bytes are available to read from an input stream and when an output stream signals that it's ready to accept bytes. For these two events, the delegate sends the stream the appropriate message—`read:maxLength:` or `write:maxlength:`, depending on type of stream—to get the bytes from the stream or to put bytes on the stream.

`NSStream` is built on the `CFStream` layer of Core Foundation. This close relationship means that the concrete subclasses of `NSStream`, `NSOutputStream` and `NSInputStream`, are toll-free bridged with their Core Foundation counterparts `CFWriteStream` and `CFReadStream`. Although there are strong similarities between the Cocoa and Core Foundation stream APIs, their implementations are not exactly coincident. The Cocoa stream classes use the delegation model for asynchronous behavior (assuming run-loop scheduling) while Core Foundation uses client callbacks. The Core Foundation stream types sets the client (termed a context in Core Foundation) differently than the NSStream sets the delegate; calls to set the delegate should not be mixed with calls to set the context. Otherwise you can freely intermix calls from the two APIs in your code.

Despite their strong similarities, `NSStream` does give you a major advantage over `CFStream`. Because of its Objective-C underpinnings, it is extensible. You can subclass `NSStream`, `NSInputStream`, or `NSOutputStream` to customize stream attributes and behavior. For example, you could create an input stream that maintains statistics on the bytes it reads; or you could make a `NSStream` subclass whose instances can seek through their stream, putting back bytes that have been read. `NSStream` has its own set of required overrides, as do `NSInputStream` and `NSOutputStream`. See the reference documentation for `NSStream`, `NSInputStream`, and `NSOutputStream` for details on subclassing these classes.

Reading From Input Streams

In Cocoa, reading from an `NSInputStream` instance consists of several steps:

1. Create and initialize an instance of `NSInputStream` from a source of data.

2. Schedule the stream object on a run loop and open the stream.

3. Handle the events that the stream object reports to its delegate.

4. When there is no more data to read, dispose of the stream object.

The following discussion goes into each of these steps in more detail.

> **Note:** The examples in this document show the strategy of scheduling stream objects on run loops and setting a delegate to handle stream events. You may use polling instead of run-loop scheduling if you prefer that approach. However, run-loop scheduling with delegation is the preferred approach for various reasons (described in "Polling Versus Run-Loop Scheduling" (page 19)), and that is why it is highlighted in this document.

Preparing the Stream Object

To begin using an `NSInputStream` object you must have (after first locating, if necessary) a source of data for the stream. The source of data can be a file, an `NSData` object, or a network socket.

> **Note:** The procedure for initializing input-stream objects from network sockets is different from the procedure for the other two data sources, and is not covered in this article. To learn about initializing an `NSInputStream` instance for a network connection, see "Setting Up Socket Streams" (page 23).

The initializers and factory methods for NSInputStream allow you to create and initialize the instance from an NSData or file. Listing 1 shows an NSInputStream instance created from a file.

Listing 1   Creating and initializing an NSInputStream object

```
- (void)setUpStreamForFile:(NSString *)path {
    // iStream is NSInputStream instance variable
    iStream = [[NSInputStream alloc] initWithFileAtPath:path];
    [iStream setDelegate:self];
    [iStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
        forMode:NSDefaultRunLoopMode];
    [iStream open];
}
```

As this example shows, after you create the object you should set the delegate (more often than not to `self`). The delegate receives `stream:handleEvent:` messages from the NSInputStream object when that object is scheduled on the run loop and has stream-related events to report, such as when there are bytes on the stream to be read.

Before you open the stream to begin the streaming of data, send a `scheduleInRunLoop:forMode:` message to the stream object to schedule it to receive stream events on a run loop. By doing this, you are helping the delegate to avoid blocking when there is no data on the stream to read. If streaming is taking place on another thread, be sure to schedule the stream object on that thread's run loop. You should never attempt to access a scheduled stream from a thread different than the one owning the stream's run loop. Finally, send the `NSInputStream` instance an `open` message to start the streaming of data from the input source.

Handling Stream Events

After a stream object is sent `open`, you can find out about its status, whether it has bytes available to read, and the nature of any error with the following messages:

```
streamStatus
hasBytesAvailable
streamError
```

The returned status is an `NSStreamStatus` constant indicating that the stream is opening, reading, at the end of the stream, and so on. The returned error is an `NSError` object encapsulating information about any error that took place. (See the reference documentation for NSStream for descriptions of `NSStreamStatus` and other stream types.)

More importantly, once the stream object has been opened, it keeps sending `stream:handleEvent:` messages to its delegate until it encounters the end of the stream. These messages include a parameter with an `NSStreamEvent` constant that indicates the type of event. For `NSInputStream` objects, the most common types of events are `NSStreamEventOpenCompleted`, `NSStreamEventHasBytesAvailable`, and `NSStreamEventEndEncountered`. The delegate is typically most interested in `NSStreamEventHasBytesAvailable` events. Listing 2 illustrates a good approach for handling this type of event.

Listing 2   Handling a bytes-available event

```
- (void)stream:(NSStream *)stream handleEvent:(NSStreamEvent)eventCode {

    switch(eventCode) {
        case NSStreamEventHasBytesAvailable:
        {
            if(!_data) {
                _data = [[NSMutableData data] retain];
            }
            uint8_t buf[1024];
            unsigned int len = 0;
            len = [(NSInputStream *)stream read:buf maxLength:1024];
            if(len) {
                [_data appendBytes:(const void *)buf length:len];
                // bytesRead is an instance variable of type NSNumber.
                [bytesRead setIntValue:[bytesRead intValue]+len];
            } else {
```

```
            NSLog(@"no buffer!");
        }
        break;
    }
    // continued
```

In this implementation of `stream:handleEvent:` the delegate uses a switch statement to identify the passed-in `NSStreamEvent` constant. If the constant is `NSStreamEventHasBytesAvailable`, the delegate first lazily creates (if necessary) an `NSMutableData` object (_data) to hold the retrieved bytes. Then it declares a buffer of a certain size (1024 bytes, in this case) and invokes the stream object's `read:maxLength:` method, which fills up the buffer with the specified number of bytes. If the read operation successfully fetched bytes from the stream, the delegate appends these bytes to the `NSMutableData` object.

There is no firm guideline on how many bytes to read at one time. Although it may be possible to read all the data in the stream in one event, this depends on the length of the stream (that is, the number of bytes in it) as well as the behavior of the kernel, including device and socket characteristics. The best approach is to use some reasonable buffer size, such as 512 bytes, one kilobyte (as in the example above), or a page size (four kilobytes).

When the `NSInputStream` object experiences errors processing the stream, it stops streaming and notifies its delegate with a `NSStreamEventErrorOccurred`. The delegate should handle the error in its `stream:handleEvent:` method as described in "Handling Stream Errors" (page 21).

Disposing of the Stream Object

When an `NSInputStream` object reaches the end of a stream, it sends the delegate a `NSStreamEventEndEncountered` event in a `stream:handleEvent:` message. The delegate should dispose of the object by doing the mirror-opposite of what it did to prepare the object. In other words, it should first close the stream object, remove it from the run loop, and finally release it. Listing 3 gives an example of how you might do this.

Listing 3   Closing and releasing the NSInputStream object

```
- (void)stream:(NSStream *)stream handleEvent:(NSStreamEvent)eventCode
{
    switch(eventCode) {
        case NSStreamEventEndEncountered:
        {
            [stream close];
            [stream removeFromRunLoop:[NSRunLoop currentRunLoop]
                forMode:NSDefaultRunLoopMode];
            [stream release];
            stream = nil; // stream is ivar, so reinit it
            break;
        }
        // continued ...
    }
}
```

Disposing of the Stream Object

Writing To Output Streams

Using an `NSOutputStream` instance to write to an output stream requires several steps:

1. Create and initialize an instance of `NSOutputStream` with a repository for the written data. Also set a delegate.

2. Schedule the stream object on a run loop and open the stream.

3. Handle the events that the stream object reports to its delegate.

4. If the stream object has written data to memory, obtain the data by requesting the `NSStreamDataWrittenToMemoryStreamKey` property.

5. When there is no more data to write, dispose of the stream object.

The following discussion goes into each of these steps in more detail.

> **Note:** The examples in this document show the strategy of scheduling stream objects on run loops and setting a delegate to handle stream events. You may use polling instead of run-loop scheduling if you prefer that approach. However, run-loop scheduling with delegation is the preferred approach for various reasons (described in "Polling Versus Run-Loop Scheduling" (page 19)), and that is why it is highlighted in this document.

Preparing the Stream Object

To begin using an `NSOutputStream` object you must specify a destination for the data written to the stream. The destination for an output-stream object can be a file, a C buffer, application memory, or a network socket.

> **Note:** The procedure for initializing output-stream objects from network sockets is different from the procedure for the other data destinations, and is not covered in this article. To learn about initializing an `NSOutputStream` instance for a network connection, see "Setting Up Socket Streams" (page 23).

The initializers and factory methods for `NSOutputStream` allow you to create and initialize the instance with a file, a buffer, or memory. Listing 1 shows the creation of an `NSOutputStream` instance that will write data to application memory.

Listing 1   Creating and initializing an NSOutputStream object for memory

```
- (void)createOutputStream {
    NSLog(@"Creating and opening NSOutputStream...");
    // oStream is an instance variable
    oStream = [[NSOutputStream alloc] initToMemory];
```

```
    [oStream setDelegate:self];
    [oStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
        forMode:NSDefaultRunLoopMode];
    [oStream open];
}
```

As the code in Listing 1 shows, after you create the object you should set the delegate (more often than not to `self`). The delegate receives `stream:handleEvent:` messages from the `NSOutputStream` object when that object has stream-related events to report, such as when the stream has space for bytes.

Before you open the stream to begin the streaming of data, send a `scheduleInRunLoop:forMode:` message to the stream object to schedule it to receive stream events on a run loop. By doing this, you are helping the delegate to avoid blocking when the stream is unable to accept more bytes. If streaming is taking place on another thread, be sure to schedule the stream object on that thread's run loop. You should never attempt to access a scheduled stream from a thread different than the one owning the stream's run loop. Finally, send the `NSOutputStream` instance an `open` message to start the streaming of data to the output container.

Handling Stream Events

After a stream object is sent `open`, you can find out about its status, whether it has space for writing data, and the nature of any error with the following messages:

```
streamStatus
hasSpaceAvailable
streamError
```

The returned status is an `NSStreamStatus` constant indicating that the stream is opening, writing, at the end of the stream, and so on. The returned error is an `NSError` object encapsulating information about any error that took place. (See the reference documentation for NSStream for descriptions of `NSStreamStatus` and other stream types.)

More importantly, once the stream object has been opened, it keeps sending `stream:handleEvent:` messages to its delegate (as long as the delegate continues to put bytes on the stream) until it encounters the end of the stream. These messages include a parameter with an `NSStreamEvent` constant that indicates the type of event. For `NSOutputStream` objects, the most common types of events are `NSStreamEventOpenCompleted`, `NSStreamEventHasSpaceAvailable`, and `NSStreamEventEndEncountered`. The delegate is typically most interested in `NSStreamEventHasSpaceAvailable` events. Listing 2 illustrates one approach you could take to handle this type of event.

Listing 2   Handling a space-available event

```
- (void)stream:(NSStream *)stream handleEvent:(NSStreamEvent)eventCode
{
    switch(eventCode) {
        case NSStreamEventHasSpaceAvailable:
        {
            uint8_t *readBytes = (uint8_t *)[_data mutableBytes];
            readBytes += byteIndex; // instance variable to move pointer
            int data_len = [_data length];
            unsigned int len = ((data_len - byteIndex >= 1024) ?
                1024 : (data_len-byteIndex));
```

```
            uint8_t buf[len];
            (void)memcpy(buf, readBytes, len);
            len = [stream write:(const uint8_t *)buf maxLength:len];
            byteIndex += len;
            break;
        }
        // continued ...
    }
}
```

In this implementation of `stream:handleEvent:` the delegate uses a switch statement to identify the passed-in `NSStreamEvent` constant. If the constant is `NSStreamEventHasSpacesAvailable`, the delegate gets the bytes held by a `NSMutableData` object (`_data`) and advances the pointer for the current write operation. It next determines the byte capacity of the impending write operation (1024 or the remaining bytes to write), declares a buffer of that size, and copies that amount of data to the buffer. Next the delegate invokes the output-stream object's `write:maxLength:` method to put the buffer's contents onto the output stream. Finally it advances the index used to advance the `readBytes` pointer for the next operation.

If the delegate receives an `NSStreamEventHasSpaceAvailable` event and does not write anything to the stream, it does not receive further space-available events from the run loop until the `NSOutputStream` object receives more bytes. When this happens, the run loop is restarted for space-available events. If this scenario is likely in your implementation, you can have the delegate set a flag when it *doesn't* write to the stream upon receiving an `NSStreamEventHasSpaceAvailable` event. Later, when your program has more bytes to write, it can check this flag and, if set, write to the output-stream instance directly.

There is no firm guideline on how many bytes to write at one time. Although it may be possible to write all the data to the stream in one event, this depends on external factors, such as the behavior of the kernel and device and socket characteristics. The best approach is to use some reasonable buffer size, such as 512 bytes, one kilobyte (as in the example above), or a page size (four kilobytes).

When the `NSOutputStream` object experiences errors writing to the stream, it stops streaming and notifies its delegate with a `NSStreamEventErrorOccurred`. The delegate should handle the error in its `stream:handleEvent:` method as described in "Handling Stream Errors" (page 21).

Disposing of the Stream Object

When an `NSOutputStream` object concludes writing data to an output stream, it sends the delegate a `NSStreamEventEndEncountered` event in a `stream:handleEvent:` message. At this point the delegate should dispose of the stream object by doing the mirror-opposite of what it did to prepare the object. In other words, it should first close the stream object, remove it from the run loop, and finally release it. Furthermore, if the destination for the `NSOutputStream` object is application memory (that is, you created the instance using `initToMemory` or the factory method `outputStreamToMemory`), you might now want to retrieve the data held in memory. Listing 3 illustrates how you might do all of these things.

Listing 3   Closing and releasing the NSInputStream object

```
- (void)stream:(NSStream *)stream handleEvent:(NSStreamEvent)eventCode
{
    switch(eventCode) {
        case NSStreamEventEndEncountered:
        {
            NSData *newData = [oStream propertyForKey:
```

```
            NSStreamDataWrittenToMemoryStreamKey];
        if (!newData) {
            NSLog(@"No data written to memory!");
        } else {
            [self processData:newData];
        }
        [stream close];
        [stream removeFromRunLoop:[NSRunLoop currentRunLoop]
            forMode:NSDefaultRunLoopMode];
        [stream release];
        oStream = nil; // oStream is instance variable
        break;
    }
    // continued ...
    }
}
```

You get the stream data written to memory by sending the `NSOutputStream` object a `propertyForKey:` message, specifying a key of `NSStreamDataWrittenToMemoryStreamKey` The stream object returns the data in an `NSData` object.

Polling Versus Run-Loop Scheduling

A potential problem with stream processing is blocking. A thread that is writing to or reading from a stream might have to wait indefinitely until there is (respectively) space on the stream to put bytes or bytes on the stream that can be read. In effect, the thread is at the mercy of the stream, and that can spell trouble for an application. Blocking can especially be a problem with socket streams because they are dependent on responses from a remote host.

With Cocoa streams you have two ways to handle stream events:

- **Run-loop scheduling**. You schedule a stream object on a run loop so that the delegate receives messages reporting stream-related events only when blocking is unlikely to take place. For read and write operations, the pertinent `NSStreamEvent` constants are `NSStreamHasBytesAvailable` and `NSStreamHasSpaceAvailable`.

- **Polling**. In a closed loop broken only at the end of the stream or upon error, you keep asking the stream object if it has (for read streams) bytes available to read or (for write streams) space available for writing. The pertinent methods are `hasBytesAvailable` (NSInputStream) and `hasSpaceAvailable` (NSOutputStream).

Run-loop scheduling is almost always preferable over polling, and that is why the code examples in "Reading From Input Streams" (page 11) and "Writing To Output Streams" (page 15) exclusively show the use of run loops. With polling, your program is locked in a tight loop, waiting for stream events that might or might not be imminent. With run-loop scheduling, your program can go off and do other things, knowing that it will be notified when there is a stream event to handle. Moreover, run loops save you from having to manage state and are more efficient than polling. Polling is also CPU-intensive; there are other things you can be doing with your processing time.

That said, there can be situations where polling is a viable option. For example, if you are porting legacy code, you might choose to use polling because it is better suited the threading model in the legacy code. Listing 1 illustrates a method that writes data to an output stream using polling.

Listing 1   Writing to an output stream using polling

```
- (void)createNewFile {
    oStream = [[NSOutputStream alloc] initToMemory];
    [oStream open];
    uint8_t *readBytes = (uint8_t *)[data mutableBytes];
    uint8_t buf[1024];
    int len = 1024;

    while (1) {
        if (len == 0) break;
        if ( [oStream hasSpaceAvailable] ) {
        (void)strncpy(buf, readBytes, len);
        readBytes += len;
        if ([oStream write:(const uint8_t *)buf maxLength:len] == -1) {
            [self handleError:[oStream streamError]];
            break;
        }
```

```
        [bytesWritten setIntValue:[bytesWritten intValue]+len];
        len = (([data length] - [bytesWritten intValue] >= 1024) ? 1024 :
            [data length] - [bytesWritten intValue]);
        }
    }
    NSData *newData = [oStream propertyForKey:
        NSStreamDataWrittenToMemoryStreamKey];
    if (!newData) {
        NSLog(@"No data written to memory!");
    } else {
        [self processData:newData];
    }
    [oStream close];
    [oStream release];
    oStream = nil;
}
```

It should be pointed out that neither the polling nor run-loop scheduling approaches are airtight defenses against blocking. If the NSInputStream `hasBytesAvailable` method or the NSOutputStream `hasSpaceAvailable` method returns `NO`, it means in both cases that the stream definitely has no available bytes or space. However, if either of these methods returns `YES`, it can mean that there is available bytes or space *or* that the only way to find out is to attempt a read or a write operation (which could lead to a momentary block). The `NSStreamEventHasBytesAvailable` and `NSStreamEventHasSpaceAvailable` stream events have identical semantics.

Handling Stream Errors

Occasionally, and especially with sockets, streams can experience errors that prevent further processing of stream data. Generally, errors indicate the absence of something at one end of a stream, such as the crash of a remote host or the deletion of a file being streamed. There is a little that a client of a stream can do when most errors occur except report the error to the user. Although a stream object that has reported an error can be queried for state before it is closed, it cannot be reused for read or write operations.

The `NSStream` and `NSOutputStream` classes inform you if an error occurred in several ways:

- If the stream object is scheduled on a run loop, the object reports a `NSStreamEventErrorOccurred` event to its delegate in a `stream:handleEvent:` message.

- At any time, the client can send a `streamStatus` message to a stream object and see if it returns `NSStreamStatusError`.

- If you attempt to write to an `NSOutputStream` object by sending it `write:maxLength:` and it returns -1, a write error has occurred.

Once you have determined that a stream object experienced an error, you can query the object with a `streamError` message to get more information about the error (in the form of an `NSError` object). Next, inform the user about the error. Listing 1 shows how the delegate of a run loop-scheduled stream object might handle an error.

Listing 1   Handling stream errors

```
- (void)stream:(NSStream *)stream handleEvent:(NSStreamEvent)eventCode {
    NSLog(@"stream:handleEvent: is invoked...");

    switch(eventCode) {
        case NSStreamEventErrorOccurred:
        {
            NSError *theError = [stream streamError];
            NSAlert *theAlert = [[NSAlert alloc] init]; // modal delegate releases
            [theAlert setMessageText:@"Error reading stream!"];
            [theAlert setInformativeText:[NSString stringWithFormat:@"Error %i: %@",
                [theError code], [theError localizedDescription]]];
            [theAlert addButtonWithTitle:@"OK"];
            [theAlert beginSheetModalForWindow:[NSApp mainWindow]
                modalDelegate:self
                didEndSelector:@selector(alertDidEnd:returnCode:contextInfo:)
                contextInfo:nil];
            [stream close];
            [stream release];
            break;
        }
        // continued ....
    }
}
```

For some errors, you can attempt to do more than inform the user. For example, if you try to set an SSL security level on a socket connection but the remote host is not secure, the stream object will report an error. You can then release the old stream object and create a new one for a non-secure socket connection.

Setting Up Socket Streams

You can use the CFStream API to establish a socket connection and, with the stream object (or objects) created as a result, send data to and receive data from a remote host. You can also configure the connection for security.

Basic Procedure

The NSStream class does not support connecting to a remote host on iOS. CFStream *does* support this behavior, however, and once you have created your streams with the CFStream API, you can take advantage of the toll-free bridge between CFStream and NSStream to cast your CFStreams to NSStreams. Just call the `CFStreamCreatePairWithSocketToHost` function, providing a host name and a port number, to receive both a `CFReadStreamRef` and a `CFWriteStreamRef` for the given host. You can then cast these objects to an NSInputStream and an NSOutputStream and proceed.

Listing 1 illustrates the use of `CFStreamCreatePairWithSocketToHost`. This example shows the creation of both a `CFReadStreamRef` object and a `CFWriteStreamRef` object. If you want to receive only one of these objects, just specify `NULL` as the parameter value for the unwanted object.

Listing 1   Setting up a network socket stream

```
- (IBAction)searchForSite:(id)sender
{
    NSString *urlStr = [sender stringValue];
    if (![urlStr isEqualToString:@""]) {
        NSURL *website = [NSURL URLWithString:urlStr];
        if (!website) {
            NSLog(@"%@ is not a valid URL");
            return;
        }

        CFReadStreamRef readStream;
        CFWriteStreamRef writeStream;
        CFStreamCreatePairWithSocketToHost(NULL, (CFStringRef)[website host],
80, &readStream, &writeStream);

        NSInputStream *inputStream = (NSInputStream *)readStream;
        NSOutputStream *outputStream = (NSOutputStream *)writeStream;
        [inputStream setDelegate:self];
        [outputStream setDelegate:self];
        [inputStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
forMode:NSDefaultRunLoopMode];
        [outputStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
forMode:NSDefaultRunLoopMode];
        [inputStream open];
        [outputStream open];
    }
```

```
    }
```

If the socket connection fails, then one or both of the requested `CFReadStreamRef` and `CFWriteStreamRef` objects are `NULL`. Once you have cast the CFStreams to NSStreams, set the delegate, schedule the stream on a run loop, and open the stream as usual. The delegate should begin to receive stream-event messages (`stream:handleEvent:`). See "Reading From Input Streams" (page 11) and "Writing To Output Streams" (page 15) for more information.

Securing and Configuring the Connection

Before you open a stream object, you might want to set security and other features for the connection to the remote host (which might be, for example, an HTTPS server). `NSStream` defines properties that affect the security of TCP/IP socket connections in two ways:

- Secure Socket Layer (SSL).

    A security protocol using digital certificates to provide data encryption, server authentication, message integrity, and (optionally) client authentication for TCP/IP connections.

- SOCKS proxy server.

    A server that sits between a client application and a real server over a TCP/IP connection. It intercepts requests to the real server and, if it cannot fulfill them from a cache of recently requested files, forwards them to the real server. SOCKS proxy servers help improve performance over a network and can also be used to filter requests.

For SSL security, `NSStream` defines various security-level properties (for example, `NSStreamSocketSecurityLevelSSLv2`). You set these properties by sending `setProperty:forKey:` to the stream object using the key `NSStreamSocketSecurityLevelKey`, as in this sample message:

```
[inputStream setProperty:NSStreamSocketSecurityLevelTLSv1
forKey:NSStreamSocketSecurityLevelKey];
```

You must set the property before you open the stream. Once it opens, it goes through a handshake protocol to find out what level of SSL security the other side of the connection is using. If the security level is not compatible with the specified property, the stream object generates an error event. However, if you request a negotiated security level (`NSStreamSocketSecurityLevelNegotiatedSSL`), the security level becomes the highest that both sides of the connection can implement. Still, if you try to set an SSL security level when the remote host is not secure, an error is generated.

To configure a SOCKS proxy server for a connection, you need to construct a dictionary with keys of the form `NSStreamSOCKSProxyNameKey` (for example, `NSStreamSOCKSProxyHostKey`). The value of each key is the SOCKS proxy setting that *Name* refers to. Then using `setProperty:forKey:`, set the dictionary as the value of the `NSStreamSOCKSProxyConfigurationKey`.

Initiating an HTTP Request

If you are opening a connection to an HTTP server (that is, a website), then you may have to initiate a transaction with that server by sending it an HTTP request. A good time to make this request is when the delegate of the `NSOutputStream` object receives a `NSStreamEventHasSpaceAvailable` event via a `stream:handleEvent:` message. "Making an HTTP GET request" shows the delegate creating an HTTP GET request and writing it to the output stream, after which it immediately closes the stream object.

Listing 2   Making an HTTP GET request

```
- (void)stream:(NSStream *)stream handleEvent:(NSStreamEvent)eventCode {
    NSLog(@"stream:handleEvent: is invoked...");

    switch(eventCode) {
        case NSStreamEventHasSpaceAvailable:
        {
            if (stream == oStream) {
                NSString * str = [NSString stringWithFormat:
                    @"GET / HTTP/1.0\r\n\r\n"];
                const uint8_t * rawstring =
                    (const uint8_t *)[str UTF8String];
                [oStream write:rawstring maxLength:strlen(rawstring)];
                [oStream close];
            }
            break;
        }
        // continued ...
    }
}
```

Document Revision History

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

Date	Notes
2009-12-16	Updated code listings in the Setting Up Socket Streams chapter.
2009-08-28	Added links to related concepts.
2009-05-06	Added a missing comment to a code sample.
2008-10-15	Fixed broken links.
2006-10-03	Fixed a broken link.
2006-04-04	Changed event in code listing on writing to a network stream to NSStreamEventHasSpaceAvailable.
2005-07-07	Fixed bugs and changed title from "Streams."
2004-07-21	Fixed bug in code example (Radar 3597799).
2004-02-20	First version of *Streams*.