ネットワーキング プログラミング トピックス

Developer

目次

リスト

概要

Important: この章には未確定の事項が一部記載されています。技術的に間違いないことは確認しておりますが、最終的なものではありません。この段階で公開したのは、ここで説明する技術やプログラミングインターフェイスを採用する一助にしていただくためです。内容は変わる可能性があるので、ソフトウェアを実装する際には、正式な資料が出た段階で確認してください。資料の更新については、AppleデベロッパWebサイトでご案内します。該当する参照ライブラリのページで、「Documents」欄に資料名を入力してください。

この資料は、特殊なネットワーク処理を実装する場合に読むべき記事を集めたものです。Developer Libraryに収録した、プログラミング上の話題を集めた他の資料と同じく、ネットワーク処理の考え方はよく理解している読者を想定しています。

Important: 大部分の開発者は、この資料を読む必要も、各記事に説明されているネットワーク処理を実装することもないでしょう。あらかじめ『*Networking Overview*』を読んで、各記事で説明しているような作業が必要になるのはどのような場合か、見当をつけてください。

この文書の使い方

このドキュメントは次の項目で構成されています。

- "ソケットやソケットストリームの使い方"（6 ページ） — ソケットやストリームを使って、低レベルのネットワーク処理（POSIX層からFoundation層まで）を実装する方法を説明します。クライアント側、サーバ側の両方について触れています。
- "DNSによるホスト名の解決"（23 ページ） — DNSを使ってホスト名を解決する方法を、よくある落とし穴の回避方法を交えながら説明します。
- "TLSにおける信頼の連鎖に沿った検証処理を安全に改変する方法"（26 ページ） — TLS（Transport Layer Security）の「信頼の連鎖」に沿った検証処理を、セキュリティ上の危険を冒すことなく、安全に改変する手順を解説します。（CFStream APIやNSStream APIによる）TCPストリームと、（NSURLConnection APIによる）URLリクエストの、両方について説明します。

各記事は、該当する処理を実装しようとする、開発者を想定してまとめたものです。

必要事項

次の資料を読んでいるか、相当する内容を理解している読者を想定します。

- 『*Networking Overview*』— ネットワークソフトウェアの基本的な動作原理や、陥りがちな罠の回避方法を説明しています。
- 『*Networking Concepts*』— ソケットベースのネットワーク処理の基本的な考え方を説明しています。

ソケットやソケットストリームの使い方

この記事では、ソケットやソケットストリームの扱い方を、POSIX層からFoundation層までの各レベルについて説明します。

> **Important:** この記事では、ソケットを使って接続し、プログラムで完全に制御できるようにする方法を説明します。多くの場合、`NSURLConnection`など高レベルのAPIの方が扱いやすいでしょう。これについては、『*Networking Overview*』を参照してください。
>
> この記事で取り上げるAPIを使うのは、CocoaやCore Foundationに装備されていないプロトコルを実装する場合に限ってください。

ネットワーク処理のどのレベルでも、ソフトウェアはクライアント側（他のアプリケーションに接続するプログラム）とサーバ側（他のアプリケーションから接続されるプログラム）の2つに分かれます。高いレベルで見れば、両者の境界は明確です。高レベルAPIを使って実装するプログラムの多くは、純粋にクライアント側です。しかし低レベルで見ると、その境界が曖昧なことも少なくありません。

ソケットやストリームを使うプログラミングは一般に、次のいずれかに分類されます。

- パケットベースの通信 - 同時に1つのパケットのみを対象として処理するプログラム。入ってくるパケットを待ち受け、パケットを送ってレスポンスします。

  この場合、クライアントとサーバの違いは、受け渡しするパケットの中身と、そして（おそらく）各プログラムがデータに対して行う処理だけです。ネットワーク処理の部分は同等です。
- ストリームベースのクライアント - TCPを使い、方向がそれぞれ異なる2つの連続するバイトストリームとして、データを受け渡しするプログラム。

  ストリームベースの通信の場合、クライアントとサーバにはより明確な違いがあります。実際にデータを処理する部分は似ていますが、最初に通信チャンネルを構築する方法はまったく異なります。

この章では、以上の区分に基づき、次の各節に分けて説明します。

- “APIファミリを選択する” （7 ページ） — ネットワーク処理の実装に用いるAPIファミリの選び方を説明します。
- “TCPベースのクライアントを開発する” （8 ページ） — 既存のサーバやサービスに向けて接続する、外向きTCP接続の方法を説明します。

- “TCPベースのサーバを開発する”（11 ページ） — サーバやサービスが、内向きのTCP接続要求を待ち受ける方法を説明します。
- “パケットベースのソケットを操作する”（19 ページ） — TCP以外のプロトコル（UDPなど）の扱い方を説明します。

APIファミリを選択する

ソケットベースの接続に使うAPIは、他のホストに接続するのか、それとも他のホストからの接続を待ち受けるのかによって異なります。また、TCPかそれ以外のプロトコルかによっても違います。いくつか考慮するべき事項を以下に示します。

- OS Xの場合、Apple以外のプラットフォームで動作するプログラムと共通のネットワーク処理コードが既にあれば、POSIX Cのネットワーク処理APIを使い、そのまま（別スレッドで）使い続けて構いません。Core Foundation層やCocoa（Foundation層）の実行ループをベースとしているならば、Core Foundation層の`CFStream` APIを使って、POSIXネットワーク処理コードを、メインスレッド上のアーキテクチャ全体に統合できます。あるいは、Grand Central Dispatch（GCD）を使っている場合、ソケットをディスパッチ源として追加することも可能です。

  iOSの場合、POSIXのネットワーク処理はお勧めしません。移動体通信やオンデマンドVPNをアクティブ化することはないからです。したがって一般に、ネットワーク処理コードを他のデータ処理機能と切り離し、高レベルAPIを使って実装し直してください。

  > **注意:** POSIX方式でネットワーク処理コードを記述する場合、そのAPIはプロトコルの詳細を知らないことを頭に入れておいてください（IPv4とIPv6の違いを吸収するコードも記述しなければなりません）。名前ではなくIPを指定して接続する方式なので、高レベルAPIと同様の処理性能や頑健性を実現するためには、多くの作業が必要です。既存のPOSIX方式のネットワーク処理コードを再利用するかどうか判断する前に、『*Networking Overview*』の“Avoid Resolving DNS Names Before Connecting to a Host” in *Networking Overview* を読んでおいてください。

- あるポートでTCP接続を待ち受け、あるいはTCP以外の接続を待ち受けるデーモンやサービスを実装したい場合は、POSIX層またはCore Foundation層（`CFSocket`）の、Cで実装されたネットワーク処理APIを使います。
- Objective-Cによるクライアント側のコードには、Objective-Cで実装された、Foundation層のネットワーク処理APIを使います。Foundation層には、URLによる接続、ソケットストリーム、ネットワークサービス、その他のネットワーク処理を管理する、高レベルのクラスが定義されています。これはOS XやiOSの、主としてUIに関与しない部分を実装したObjective-Cのフレームワークでもあり、実行ループ、文字列処理、コレクションオブジェクト、ファイルアクセスなどのルーチンがあります。

- クライアント側のコードをCで実装する場合は、Core Foundation層のCによるネットワーク処理APIを使います。Core Foundationフレームワーク、CFNetworkフレームワークは、OS X用およびiOS用の、Cで実装されたフレームワークです。いずれも関数や構造体の定義があり、その上にFoundation層のネットワーク処理クラスが構築されています。

  注意: OS Xの場合、CFNetworkはCore Servicesフレームワークの一部です。一方、iOSの場合、CFNetworkは独立したフレームワークになっています。

TCPベースのクライアントを開発する

外向きの接続方法は、開発に用いる言語、接続方式（TCP、UDPなど）、他の（MacやiOS以外の）プラットフォームとコードを共有するか否か、によって異なります。

- `NSStream`を使う方法（外向きの接続、Objective-Cで記述）。

  特定のホストに接続する場合は、`CFHost`オブジェクトを生成（`NSHost`ではないことに注意。これはフレームワークの相互乗り入れ不可）し、`CFStreamCreatePairWithSocketToHost`または`CFStreamCreatePairWithSocketToCFHost`で当該ホストおよびポートに接続するソケットを開き、`CFStream`オブジェクトの組（送信用と受信用）を対応づけます。次に、このストリームを`NSStream`オブジェクトにキャストします。

  また、`CFNetServiceRef`オブジェクトを引数として`CFStreamCreatePairWithSocketToNetService`関数を呼び出すことにより、Bonjourサービスに接続することも可能です。詳しくは、『*Networking Overview*』の“Discovering and Advertising Network Services”を参照してください。

  注意: `NSNetService`の`getStreamsToHost:port:inputStream:outputStream:`メソッドは、iOSでは使えません。OS Xの場合も、性能に問題があるのでお勧めできません。具体的に言うと、`NSNetService`を使うためには、`NSHost`のインスタンスを生成する必要があります。その際、検索は同期的に実行しなければなりません。そのため、アプリケーションのメインスレッド上で`NSHost`オブジェクトを構築するのは、安全とは言えないのです。詳細については、『*NSNetService and Automatic Reference Counting (ARC)*』を参照してください。

- `CFStream`を使う方法（外向きの接続、Cで記述）。

  Objective-Cで記述できない場合は、`CFStream` APIを使ってください。`CFSocket`に比べて、Core Foundation層の他のAPIを組み込むのが容易です。また、低レベルのAPIと違い、（可能な場合）iOS上で移動体通信ハードウェアを有効にする働きもあります。

`CFStreamCreatePairWithSocketToHost`または`CFStreamCreatePairWithSocketToCFHost`で、特定のホストおよびポートに接続するソケットを開き、`CFStream`オブジェクトの組（送信用と受信用）を対応づけることができます。

また、`CFStreamCreatePairWithSocketToNetService`関数で、Bonjourサービスに接続することも可能です。詳しくは、『*Networking Overview*』の“Discovering and Advertising Network Services”を参照してください。

- POSIX APIを使う方法（プラットフォームをまたがる可搬性が重要な場合）。

  もっぱらOS XやiOSのみで動作するネットワーク処理コードであれば、一般に、POSIXのネットワーク処理システムコールは使わないでください。高レベルAPIに比べて記述量が増えてしまいます。しかし、他のプラットフォームとコードを共有する場合は、同じコードがどこででも動作するよう、POSIXネットワークAPIを使わなければなりません。

  GUIアプリケーションのメインスレッドでは、同期型のPOSIXネットワーク処理APIを実行しないでください。この場合、独立したスレッド上で実行する必要があります。

  > **注意:** POSIXのネットワーク処理APIでは、iOSの移動体通信（無線通信）がアクティブになりません。そのため、iOSでは一般に、POSIXのAPIはお勧めできません。

以下、`NSStream`の使い方を説明します。特に注記しない限り、`CFStream` APIにも同じような名前の関数があり、動作も似ています。

POSIXのソケットAPIについては、『UNIX Socket FAQ』（http://developerweb.net/）を参照してください。

接続を確立する

一般に、遠隔ホストへのTCP接続を確立するためには、ストリームを使う方法を推奨します。こうすれば、TCP接続に伴う面倒な問題の多くを任せてしまうことができます。たとえばホスト名を指定して接続できるほか、iOSの場合、（`CFSocket`やBSDソケットと違い）必要に応じて自動的に、デバイスの携帯電話モデムやオンデマンドVPNがアクティブになります。また、低レベルのプロトコルに比べてCocoa風のインターフェイスであり、Cocoaファイルストリーム APIとの互換性も高くなっています。

ホストとのやり取りに使う入出力ストリームの取得方法は、ホストの探索にDNS Service Discoveryを使ったかどうかによって異なります。

- 遠隔ホストのDNS名またはIPアドレスが既知であれば、Core Foundation層の読み取り/書き出し（入力/出力）用ストリームは、`CFStreamCreatePairWithSocketToHost`関数で取得できます。次に、`CFStream`と`NSStream`は相互乗り入れ可能であることを利用して、`CFReadStreamRef`オブジェクトや`CFWriteStreamRef`オブジェクトを、`NSInputStream`や`NSOutputStream`にそれぞれキャストしてください。
- `CFNetServiceBrowser`オブジェクトでネットワークサービスを探索して当該ホストを見つけたのであれば、サービスの入出力ストリームは`CFStreamCreatePairWithSocketToNetService`関数で取得できます。詳しくは、『*Networking Overview*』の"Discovering and Advertising Network Services"を参照してください。

自動参照カウントの仕組みを使っていない場合は、入出力ストリームを取得した後、直ちにこれを保持する必要があります。次にこれを`NSInputStream`および`NSOutputStream`オブジェクトにキャストし、（`NSStreamDelegate`プロトコルに準拠した）デリゲートオブジェクトを設定し、現在の実行ループ上でスケジューリングした後、`open`メソッドを実行してください。

> **注意:** 同時に複数の接続を扱う場合、ある入力ストリームがどの出力ストリームに対応するか（あるいはその逆）を追跡していなければなりません。簡単なのは、対応する2つのストリームの参照を保持する独自の「接続」オブジェクトを生成し、それぞれのデリゲートとして当該オブジェクトを設定する、という方法でしょう。

イベントの処理

`NSOutputStream`オブジェクトのデリゲートの`stream:handleEvent:`メソッドが呼び出され、その引数`streamEvent`の値が`NSStreamEventHasSpaceAvailable`であれば、`write:maxLength:`メソッドでデータを送信してください。このメソッドは、実際に書き出したバイト数（エラーの場合は負の数）を返します。送信しようとしたバイト数よりも少なければ、残ったデータをキューに入れておきます。次に`NSStreamEventHasSpaceAvailable`イベントが発生し、デリゲートメソッドが呼び出された時点で、残りを送信することになります。エラーが発生した場合は`streamError`を実行して原因を調べます。

`NSInputStream`オブジェクトのデリゲートの`stream:handleEvent:`メソッドが呼び出され、その引数`streamEvent`の値が`NSStreamEventHasBytesAvailable`であれば、入力ストリームがデータを受け取ったことを表し、`read:maxLength:`メソッドで読み込むことができます。このメソッドは実際に読み込んだバイト数（エラーの場合は負の数）を返します。

必要なバイト数よりも少なければ、データをキューに入れておき、追加データが届いた旨のイベントを受け取るまで待たなければなりません。エラーが発生した場合は`streamError`を実行して原因を調べます。

接続の他端側が接続を閉じた場合、次のいずれかが起こります。

- 接続デリゲートの`stream:handleEvent:`メソッドが呼び出され、その引数`streamEvent`の値が`NSStreamEventHasBytesAvailable`である。このストリームからデータを読み込もうとしても、長さは`0`となります。
- 接続デリゲートの`stream:handleEvent:`メソッドが呼び出され、その引数`streamEvent`の値が`NSStreamEventEndEncountered`である。

上記のいずれかが起こった場合、デリゲートメソッドはファイル末尾に到った旨を検出し、後始末をしなければなりません。

接続を閉じる

接続を閉じたい場合は、実行ループから外し（スケジューリングを解除し）、接続のデリゲートを`nil`とし（デリゲートの保持を解除）、対応する2つのストリームを`close`メソッドで閉じた上で、（ARCを使っていない場合は）ストリームを解放し、または（ARCを使っている場合は）ストリームとして`nil`を設定してください。その結果、通常は、内部的に保持しているソケット接続も閉じられます。しかし次のような場合、意識的に閉じなければなりません。

- ストリームの`setProperty:forKey:`メソッドで、`kCFStreamPropertyShouldCloseNativeSocket`の値を`kCFBooleanFalse`と設定していた場合。
- 既存のBSDソケットをもとに、`CFStreamCreatePairWithSocket`関数でストリームを生成した場合。

  このようなストリームは、通常、内部的に保持しているソケットを閉じるようになっていません。しかし、`setProperty:forKey:`メソッドで`kCFStreamPropertyShouldCloseNativeSocket`の値を`kCFBooleanTrue`と設定することにより、自動的に閉じるようにすることも可能です。

詳しい情報の入手先

さらに詳しくは、『*Stream Programming Guide*』の“Setting Up Socket Streams”、『*Using NSStreams For A TCP Connection Without NSHost*』、あるいはサンプルコードプロジェクト『*SimpleNetworkStreams*』や『*RemoteCurrency*』を参照してください。

TCPベースのサーバを開発する

先に説明したように、接続が確立してしまえば、以後の処理はサーバもクライアントもほとんど同じです。一番の違いは、クライアントは外向きに接続するのに対し、サーバは**待ち受けソケット**を生成して待機し、接続要求があればそれを受理する、という点です。受理してしまえば、その接続は、クライアント側と同じように動作します。

サーバ側で用いるAPIは、他の（MacやiOS以外の）プラットフォームとコードを共有するか否か、によって異なります。内向きのネットワーク接続要求を待ち受けるAPIは、Core Foundation層のソケットAPIと、POSIX層の（BSD）ソケットAPIの2種類しかありません。これより高レベルのAPIに、内向きの接続要求を受理する機能はないのです。

- OS XおよびiOS専用であれば、POSIX層のネットワーク処理システムコールを使ってネットワークソケットを用意してください。このソケットを、GCDまたは`CFSocket`を使って、実行ループに組み込みます。
- Apple以外のプラットフォームでも動作する、可搬性のあるコードを記述したい場合は、POSIX層のネットワーク処理システムコールを使い、POSIX層ベースの実行ループに組み込みます（`select`）。

  もっぱらOS XやiOSのみで動作するネットワーク処理コードであれば、一般に、POSIXのネットワーク処理システムコールは使わないでください。高レベルAPIに比べて記述量が増えてしまいます。しかし、他のプラットフォームとコードを共有する場合は、同じコードがどこででも動作するよう、POSIXネットワークAPIを使わなければなりません。
- 一般的なソケット通信に、`NSSocketPort`や`NSFileHandle`は使いません。詳しくは、『*Networking Overview*』の“Do Not Use NSSocketPort (OS X) or NSFileHandle for General Socket Communication” in *Networking Overview*を参照してください。

以下、接続要求を待ち受けるAPIの使い方を説明します。

接続要求を待ち受ける（Core Foundation層）

Core Foundation層のAPIで接続要求を待ち受ける手順を以下に示します。

1. 適切なincludes文を追加します。

```
#include <CoreFoundation/CoreFoundation.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

2. ソケットオブジェクトを、`CFSocketCreate`関数または`CFSocketCreateWithNative`関数で生成します（`CFSocketRef`オブジェクトの形で返されます）。このとき、引数`callBackTypes`の値として`kCFSocketAcceptCallBack`を指定します。また、`CFSocketCallBack`型のコールバック関数のポインタを、引数`callout`の値として指定します。

```
CFSocketRef myipv4cfsock = CFSocketCreate(
    kCFAllocatorDefault,
```

```
    PF_INET,
    SOCK_STREAM,
    IPPROTO_TCP,
    kCFSocketAcceptCallBack, handleConnect, NULL);
CFSocketRef myipv6cfsock = CFSocketCreate(
    kCFAllocatorDefault,
    PF_INET6,
    SOCK_STREAM,
    IPPROTO_TCP,
    kCFSocketAcceptCallBack, handleConnect, NULL);
```

3. `CFSocketSetAddress`関数でソケットをバインドします。引数として、接続に用いるポートやアドレスファミリを指定した`CFData`構造体を収容する、`sockaddr`オブジェクトを渡します。

```
struct sockaddr_in sin;

memset(&sin, 0, sizeof(sin));
sin.sin_len = sizeof(sin);
sin.sin_family = AF_INET; /* アドレスファミリ */
sin.sin_port = htons(0); /* または具体的なポート番号 */
sin.sin_addr.s_addr= INADDR_ANY;

CFDataRef sincfd = CFDataCreate(
    kCFAllocatorDefault,
    (UInt8 *)&sin,
    sizeof(sin));

CFSocketSetAddress(myipv4cfsock, sincfd);
CFRelease(sincfd);

struct sockaddr_in6 sin6;

memset(&sin6, 0, sizeof(sin6));
sin6.sin6_len = sizeof(sin6);
sin6.sin6_family = AF_INET6; /* アドレスファミリ */
```

```
sin6.sin6_port = htons(0); /* または具体的なポート番号 */
sin6.sin6_addr = in6addr_any;

CFDataRef sin6cfd = CFDataCreate(
    kCFAllocatorDefault,
    (UInt8 *)&sin6,
    sizeof(sin6));

CFSocketSetAddress(myipv6cfsock, sin6cfd);
CFRelease(sin6cfd);
```

4. ソケットを実行ループに追加（登録）して待ち受けを開始します。

   まず、ソケット用の実行ループ源を、`CFSocketCreateRunLoopSource`関数で生成します。次に、この実行ループ源を`CFRunLoopAddSource`関数に渡して、ソケットを実行ループに追加します。

```
CFRunLoopSourceRef socketsource = CFSocketCreateRunLoopSource(
    kCFAllocatorDefault,
    myipv4cfsock,
    0);

CFRunLoopAddSource(
    CFRunLoopGetCurrent(),
    socketsource,
    kCFRunLoopDefaultMode);

CFRunLoopSourceRef socketsource6 = CFSocketCreateRunLoopSource(
    kCFAllocatorDefault,
    myipv6cfsock,
    0);

CFRunLoopAddSource(
    CFRunLoopGetCurrent(),
    socketsource6,
    kCFRunLoopDefaultMode);
```

これ以降、内部的に使っているBSDソケット記述子を、`CFSocketGetNative`関数で取得できます。

使い終わったソケットは、`CFSocketInvalidate`関数で閉じなければなりません。

待ち受けソケットのコールバック関数（この場合は`handleConnect`）では、引数`callbackType`の値が`kCFSocketAcceptCallBack`であること、すなわち、新たな接続要求を受理したことを確認してください。この場合、コールバック関数に渡される引数`data`は、ソケットを表す`CFSocketNativeHandle`値（整数のソケット番号）のポインタです。

新たに受理した接続を用いた送受信には、`CFStream`、`NSStream`、`CFSocket`のいずれかのAPIを使います。特にストリームベースのAPIを推奨します。

これを行うには、次のようにします。

1. ソケット用の読み込み/書き出しストリームを、`CFStreamCreatePairWithSocket`関数で生成します。
2. （Cocoa上に構築する場合）ストリームをそれぞれ、`NSInputStream`オブジェクトおよび`NSOutputStream`オブジェクトにキャストします。
3. “TCPベースのクライアントを開発する”（8 ページ）と同様にしてストリームを使います。

詳細については、『*CFSocket Reference*』を参照してください。サンプルコードプロジェクト『*RemoteCurrency*』および『*WiTap*』に使用例があります。

接続要求を待ち受ける（POSIX層のSocket API）

POSIX層のネットワーク処理も`CFSocket` APIとほとんど同じですが、実行ループ処理コードは独自に記述しなければなりません。

> **Important:** GUIアプリケーションのメインスレッドでは、POSIX層のネットワーク処理APIを実行しないでください。この場合、別のスレッド上で実行するか、GCDを利用しなければなりません。

POSIXレベルでサーバを実装する、基本的な手順を以下に示します。

1. `socket`でソケットを生成します。例：

```
int ipv4_socket = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
int ipv6_socket = socket(PF_INET6, SOCK_STREAM, IPPROTO_TCP);
```

2. ポートにバインドします。
   - ポート番号が決まっている場合はそれを使います。

- 特に決まっていない場合は、0を指定すれば、オペレーティングシステムが一時的なポート番号を割り当てるようになっています（サービスをBonjourで広告するのであれば、多くの場合、一時的なポートを使うことになります）。

例：

```
struct sockaddr_in sin;
memset(&sin, 0, sizeof(sin));
sin.sin_len = sizeof(sin);
sin.sin_family = AF_INET; // またはAF_INET6（アドレスファミリ）
sin.sin_port = htons(0);
sin.sin_addr.s_addr= INADDR_ANY;

if (bind(listen_sock, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
    // エラーを処理する。
}
```

3. 一時的なポートの場合は、`getsockname`で割り当てられたポート番号を取得します。このポートをBonjourで登録できます。例：

```
socklen_t len = sizeof(sin);
if (getsockname(listen_sock, (struct sockaddr *)&sin, &len) < 0) {
    // ここでエラーを処理
}
// ポート番号をntohs(sin.sin_port)で取得。
```

4. `listen`を実行して、当該ポートに入ってくる接続要求を待ち受けます。

これ以降の手順は、POSIX層のソケットを使って記述するか、高レベルのAPIを使うかによって異なります。

Core Foundation層でイベントを処理する

`CFSocketCreateWithNative`を呼び出します。その後は"接続要求を待ち受ける（Core Foundation層）"（12 ページ）の手順3以降に従ってください。

Grand Central Dispatchでイベントを処理する

GCDを使えば、処理を非同期に実行できるほか、イベントキューの仕組みを利用して、ソケットからいつデータを読み込むかを判断できます。GCDベースでサーバを構築する場合、待ち受けソケットを生成した後、次のように処理を進めることになります。

1. `dispatch_source_create`で待ち受けソケットのディスパッチ源を生成します。このとき、ディスパッチ源の種類として`DISPATCH_SOURCE_TYPE_READ`を指定します。
2. `dispatch_source_set_event_handler`（または`dispatch_source_set_event_handler_f`と`dispatch_set_context`）で、ソケットに接続要求が届いた時に呼び出されるハンドラを設定します。
3. （接続要求が届いた時に呼び出される）ソケットの待ち受けハンドラは、次のような処理をしなければなりません。
   - `accept`を呼び出します。この関数は、接続方法に関する情報を`sockaddr`構造体に設定し、当該接続を処理するソケットを生成して返します。

     必要ならば`ntohl(my_sockaddr_obj.sin_addr.s_addr)`を実行して、クライアントのIPアドレスを調べます。
   - `dispatch_source_create`でクライアント側ソケットのディスパッチ源を生成します。このとき、ディスパッチ源の種類として`DISPATCH_SOURCE_TYPE_READ`を指定します。
   - `setsockopt`で、ソケットの`SO_NOSIGPIPE`フラグをオンにします。
   - `dispatch_source_set_event_handler`（または`dispatch_source_set_event_handler_f`と`dispatch_set_context`）で、接続状態が変化したときに呼び出されるハンドラを設定します。
4. クライアント側ソケットハンドラでは、当該スレッド上で`read`を呼び出すブロックを引数として、`dispatch_async`または`dispatch_async_f`を呼び出して新しいデータを取得し、適切に処理します。このブロック内で、ソケットの`write`を実行してレスポンスを送ることも可能です。

イベントを処理する（純粋なPOSIXコード）

1. ファイル記述子集合（fd_set）を生成します。接続要求が届く都度、新たにソケットを生成して追加することになります。

```
fd_set incoming_connections;
memset(&incoming_connections, 0, sizeof(incoming_connections));
```

2. ネットワーク処理スレッド上で、定期的にあるアクションを実行する場合は、`select`の引数（タイムアウトの指定）として渡す、`timeval`構造体を構築します。

```
struct timeval tv;
tv.tv_sec = 1; /* タイムアウト：1秒 */
tv.tv_usec = 0; /* ミリ秒単位の指定はなし。*/
```

無理のないタイムアウト値を選ぶことが重要です。短すぎると、プロセスがあまりにも頻繁に実行され、システムが正常に動作しなくなるおそれがあります。何か特別なことがしたい場合を除き、`select`ループでは、毎秒数回程度以上ウェイクアップするべきではありません。また、iOSの場合は、定期的な実行そのものを回避できないか検討するべきでしょう。代替手段については、『*Networking Overview*』の"Avoid POSIX Sockets and CFSocket on iOS Where Possible" in *Networking Overview*を参照してください。

定期的に実行する処理がなければ`NULL`を渡します。

3. ループ内で`select`を実行します。このとき、読み取り用、書き出し用のファイル記述子を*それぞれ別個に*用意し（`FD_COPY`で複製）、引数として渡します。システムコール`select`は、ファイル記述子集合を書き換え、読み取りも書き込みもできない状態のファイル記述子はクリアするようになっています。

   引数`timeout`には、先に生成した`timeval`構造体を渡します。OS XやiOSではこの構造体を書き換えませんが、オペレーティングシステムによっては、残り時間の値で上書きすることもあります。したがって、プラットフォーム間の互換性を考慮し、`select`の呼び出しごとにこの値をリセットするようにしてください。

   引数`nfds`には、実際に使っているファイル記述子の最大値に、1を足した値を指定します。

4. ソケットからデータを読み取るときは、`FD_ISSET`マクロを実行して、ソケットに未処理のデータがあるかどうか調べながら進めます。

   データを書き出す場合も、`FD_ISSET`マクロで、ソケットに新しいデータを書き出せるだけのバッファが残っているかどうか調べながら進めます。

   読み書きするデータのキューを適切に管理してください。

POSIX層の`select`関数に代わる手段として、BSD特有の`kqueue` APIで、ソケットイベントを処理することも可能です。

詳しい情報の入手先

POSIXネットワーク処理について詳しくは、`socket`、`listen`、`FD_SET`、`select`のマニュアルページを参照してください。

パケットベースのソケットを操作する

UDPパケットを送受信する方法としては、POSIX APIと、`CFSocket`またはGCD APIを組み合わせるやり方を推奨します。その手順は次のようになります。

1. `socket`でソケットを生成します。
2. `bind`でソケットをバインドします。ポートやアドレスファミリを指定した、`sockaddr`構造体を渡します。
3. `connect`でソケットを接続します（必要な場合）。

   接続したUDPソケットは、本来の意味の接続ではないことに注意してください。もっとも、接続していないソケットと比べて、2つの利点があります。まず、メッセージを送信する都度、送信先アドレスを指定する必要がありません。また、パケットを配送できなかったとき、エラー情報を受け取れることがあります。もっとも、UDPの場合、エラー情報の配送は保証されていません。ネットワーク条件に依存し、これをアプリケーション側で制御することはできないのです。

これ以降、接続の処理方法は3通りあります。

- GCDを使って実行ループに統合する（推奨）場合、ディスパッチ源を`dispatch_source_create`で生成します。次にイベントハンドラをディスパッチ源に割り当てます。必要ならば取り消しハンドラも割り当てます。最後に、ディスパッチ源を`dispatch_resume`関数に渡して呼び出し、イベント処理を開始します。
- `CFSocket`を使って統合する方法は、若干複雑ですが、Cocoa APIと連携しやすいという利点があります。もっとも、`CFSocket`オブジェクトは単一のオブジェクトで接続を表す（POSIX層のソケットと同様）のに対し、Cocoa APIの多くは、送信と受信に別々のオブジェクトを使う、ストリームベースのAPIと相性がよい設計になっています。したがって、読み取り/書き込みストリームを扱うCocoa APIの中には、`CFSocketRef`オブジェクトとの連携が難しいものもあります。

  `CFSocket`を使う手順は次の通りです。

  1. 接続の管理に用いるオブジェクトを生成します。Objective-Cで記述する場合、これはクラスにしてもよいでしょう。Cで記述する場合は、可変（mutable）辞書など、Core Foundation層のオブジェクトでなければなりません。
  2. 当該オブジェクトを記述する、コンテキストオブジェクトを生成します。

```
CFSocketContext ctxt;


ctxt.version = 0;
ctxt.info = my_context_object;
ctxt.retain = CFRetain;
ctxt.release = CFRelease;
```

```
ctxt.copyDescription = NULL;
```

3. `CFSocketNativeHandle`オブジェクトで表されるソケットをもとに、`CFSocketCreateWithNative`で`CFSocket`オブジェクト（`CFSocketRef`）を生成します。

   このとき、引数`callBackTypes`の値は、少なくとも`kCFSocketDataCallBack`フラグを立てたものにしてください。逆に`kCFSocketAcceptCallBack`フラグはオフにします。

   また、引数`callout`の値として、コールバック関数`CFSocketCallBack`のポインタを渡します。

   例：

```
CFSocketRef connection = CFSocketCreateWithNative(kCFAllocatorDefault,
    sock,
    kCFSocketDataCallBack,
    handleNetworkData,
    &ctxt);
```

4. Core Foundation層のソケット（CFSocket）に対し、これが無効になったとき、内部的に管理しているネイティブソケットを閉じてよい旨を指定します。

```
CFOptionFlags sockopt = CFSocketGetSocketFlags(connection);


sockopt |= kCFSocketCloseOnInvalidate |
kCFSocketAutomaticallyReenableReadCallBack;
CFSocketSetSocketFlags(connection, sockopt);
```

5. ソケットのイベント源を生成し、実行ループ内で実行するようスケジューリングします。

```
CFRunLoopSourceRef socketsource = CFSocketCreateRunLoopSource(
    kCFAllocatorDefault,
    connection,
    0);


CFRunLoopAddSource(CFRunLoopGetCurrent(), socketsource,
kCFRunLoopDefaultMode);
```

新しいデータを処理できる状態になると、データハンドラコールバックが呼び出されます。その引数`callbackType`の値が`kCFSocketConnectCallBack`であれば、引数`data`の値を調べてください。これが`NULL`であれば、ホストに接続したことになるので、`CFSocketSendData`関数でデータを送信できます。

ソケットの処理が終了したら、`CFSocketInvalidate`関数で無効にしてください。

内部的に管理しているBSDソケットは、いつでも`CFSocketGetNative`関数で取得できます。

詳細については、『*CFSocket Reference*』を参照してください。『*UDPEcho*』サンプルコードプロジェクトも参考になるでしょう。

- POSIXソケットを生で使っている場合は、システムコール`select`でデータを待ち受け、`read`や`write`でI/Oの処理を行います。POSIXソケットAPIでUDPパケットを送受信する方法については、『UNIX Socket FAQ』（http://developerweb.net/）を参照してください。

ソケットストリームに対応するネイティブのソケットハンドルを取得する

ソケットベースのストリーム（`NSInputStream`、`NSOutputStream`、`CFReadStream`、`CFWriteStream`）を扱う際、その内部実装に使われているソケットハンドルが必要になることがあります。たとえば、ストリームの末端に当たるIPアドレスやポート番号を`getsockname`や`getpeername`で取得する、ソケットのオプションを`setsockopt`で設定する、などの場合に必要です。

入力ストリームに対応するネイティブのソケットハンドルは、次のようにして取得できます。

```
-(int) socknumForNSInputStream: (NSStream *)stream
{
    int sock = -1;
    NSData *sockObj = [stream propertyForKey:
                (__bridge NSString *)kCFStreamPropertySocketNativeHandle];
    if ([sockObj isKindOfClass:[NSData class]] &&
         ([sockObj length] == sizeof(int)) ) {
        const int *sockptr = (const int *)[sockObj bytes];
        sock = *sockptr;
    }
    return sock;
}
```

出力ストリームについても同様ですが、実際に取得するのはどちらか一方で構いません。同じ接続の入力ストリームと出力ストリームは、必ず同じソケットを共有するからです。

> **注意:** Core Foundation層のストリームについては、`CFReadStreamCopyProperty`、`CFDataGetLength`、`CFDataGetBytes`を使って同じことができます。

DNSによるホスト名の解決

この記事では、DNSでホスト名を解決する、適応性の高い方法を説明します。

Important: OS XやiOSの高レベルAPIの多くは、DNSホスト名を指定してホストに接続できるようになっているので、一般にこの方法を用いるとよいでしょう。同じホスト名に複数のIPアドレスが対応していることがあります。ホスト名を自分で解決する場合、実際に接続する際に、どのIPアドレスを使うか選択しなければなりません。これに対し、ホスト名を指定して接続すれば、オペレーティングシステム側が最適な方法を判断します。Bonjourサービスの場合、いつでもIPアドレスが変化しうるので、特にこの点が重要です。

しかし、名前を指定して接続できない状況もあります。IPアドレスで指定するAPIでなければ要件を満たせない場合に備えて、OS XやiOSには、DNSホスト名に対応するIPアドレスの取得手段が備わっています。この付録では、この技法について説明します。

この付録を読む前に、"Avoid Resolving DNS Names Before Connecting to a Host" in *Networking Overview* を読んでおいてください。以下の記事は読む必要がないかもしれません。

OS XやiOSにはホスト名を解決する主なAPIが3つあります。`NSHost`（OS Xのみ）、`CFHost`、POSIXのresolver APIです。

- NSHost — ホスト名を他のAPIに渡す方法としては`NSHost`を使うのが一般的ですが、`NSHost`を使って自分でアドレスを解決する方法は、一般にお勧めできません。これは同期型のAPIなので、メインスレッドで使うと深刻な性能低下につながるおそれがあるからです。このAPIは推奨しないので説明は省略します。
- CFHost — `CFHost` APIを使えば、非同期的にホスト名を解決できます。自分で名前を解決しなければならない場合の手段として推奨します。
- POSIX — POSIX層にもホスト名を解決する関数がいくつかあります。これを使うのは、非Appleプラットフォームと共有する可搬性の高いコードを記述したい、あるいは、既存のPOSIXネットワーク処理コードと組み合わせたい場合に限ってください。

ホスト名を解決する（CFHost）

`CFHost`を使ってホスト名を解決する手順を以下に示します。

1. `CFHostCreateWithName`で`CFHostRef`オブジェクトを生成します。

2. CFHostSetClientを実行します。引数として、適当なコンテキストオブジェクトと、解決できたときに呼び出されるコールバック関数を渡します。
3. CFHostScheduleWithRunLoopを呼び出して、実行ループにリゾルバをスケジューリングします。
4. CFHostStartInfoResolutionで、ホスト名の解決処理を始めるよう指示します。IPアドレスを取得したいので、第2引数にはkCFHostAddressesを指定してください。
5. コールバック関数が呼び出されるのを待ち、その中で、CFHostGetAddressingを実行して結果を取得します。この関数の戻り値は、CFDataRefオブジェクト（POSIXのsockaddr構造体）の配列です。

逆引き（IPアドレスからホスト名への変換）も同様の手順ですが、CFHostCreateWithAddressでオブジェクトを生成し、第2引数にkCFHostNamesを指定してCFHostStartInfoResolutionを実行し、CFHostGetNamesで結果を検索することになります。

ホスト名を解決する（POSIX）

POSIXシステムコールを使ってホスト名を解決する場合、これは同期呼び出しなので、GUIアプリケーションのメインスレッドでは使えないことを忘れないでください。独立したPOSIXスレッドまたはGCDタスクを生成し、そのコンテキストで実行する必要があります。

> **Important:** iOSの場合、メインスレッド上でPOSIXシステムコールを実行してホスト名を解決しようとしても、タイムアウトのとき強制終了になるおそれがあります。

POSIXでは、ホスト名を解決する3つの関数が、<netdb.h>に定義されています。

`getaddrinfo`

ホスト名を解決して得られたアドレスすべてを返します。POSIXレベルでアドレス情報を取得する方法としてはこれを推奨します。コード例がマニュアルページ（上記のリンク）に載っています。

> **Important:** 旧DNSサーバの中には、IPv6アドレスを検索しようとしたとき、エラーを返さないものがあります。POSIXのgetaddrinfo関数はこれに対応するため、IPv4アドレスのレスポンスがあれば即座にIPv6アドレスの問い合わせを取り消すようになっています。うまく接続できるまでIPv6アドレスの検索を続行したい（IPv4アドレスによる接続が成功または失敗して終わっても、すぐには検索を中止しない）場合は、CFHostなど非同期型のAPIを使ってください。

`gethostbyname`

ホスト名に対応するIPv4アドレスをひとつ返します。IPv4アドレスしか扱えないので、今後の新規開発には使わないようにしてください。

`gethostbyname2`

ホスト名に対応する、指定したアドレスファミリ（`AF_INET`、`AF_INET6`など）のアドレスをひとつ返します。

`gethostbyname`と違い、IPv4アドレスしか扱えないという問題はありませんが、複数のアドレスを試して最も高速なものを選ぶ、という処理はやはりできません。これは、既存のコードに`gethostbyname`が使われているとき、簡単に置き換えて使うことを主に意図した関数です。今後の新規開発には`getaddrinfo`を使うようお勧めします。

逆引き（IPアドレスに対応するホスト名を調べる）用には`getnameinfo`と`gethostbyaddr`があります。`getnameinfo`関数の方が適応性が高いので、これを使うようお勧めします。

詳しくは、それぞれのマニュアルページ（上記のリンク）を参照してください。

TLSにおける信頼の連鎖に沿った検証処理を安全に改変する方法

TLS（Transport Layer Security）には、ネットワーク接続の安全性を確保するため、信頼の連鎖に沿って検証処理を行う仕組みがあります。この記事では、この仕組みを改変する方法を説明します。

> **Important:** この記事は、『*Security Overview*』の"Cryptographic Services"を読むなどして、証明書、署名、信頼の連鎖について予備知識がある読者を想定しています。
>
> 大部分の実用ソフトウェアは、信頼の連鎖に沿った検証の仕組みを変更するべき*ではありません*。しかし開発過程では必要になることもよくあります。ここで説明する技法により検証の仕組みを改変した場合、誤ってデバッグ用のコードを無効にせずそのまま出荷すると、ユーザは適切に保護されないことになってしまいます。

TLS証明書を検証する場合、オペレーティングシステムは、信頼の連鎖に沿って順次、中間証明書の検証を行います。有効な証明書のみが連なっており、最終的に既知の（信頼できる）アンカ証明書に到達すれば、その証明書は有効であると判断します。そうでなければ無効ということになります。大手認証サービス企業が署名した証明書を使っている限り、その証明書は有効と判断されるはずです。

しかし、何か通常とは異なることがしたい場合、オペレーティングシステムがその証明書を有効と判断するよう、何らかの措置が必要です。通常とは異なることとしては、たとえば、ユーザ向けにクライアント証明書を作成する、複数ドメイン用のサービスをそこでは信頼されない単一の証明書で提供する、自己署名証明書を使う、IPアドレスで（ネットワーク処理スタックがホスト名を逆引きできない）ホストに接続する、などが考えられます。

大づかみにいえば、信頼の連鎖に沿ってTLS検証を実行するのは、「信頼」オブジェクト（`SecTrustRef`）です。このオブジェクトには、実施する検証処理の種類を制御する、いくつものフラグがあります。通常、このフラグを操作するべきではありませんが、その存在は認識している必要があります。さらに、信頼オブジェクトにはポリシー（`SecPolicyRef`）を設定し、TLS証明書を評価する際に用いるべきホスト名を渡せるようになっています。最後に、信頼オブジェクトは信頼できるアンカ証明書のリストを保持しており、アプリケーションはこれを書き換えることができます。

この記事はいくつかの部分に分かれています。最初に"信頼オブジェクトを操作する"（27 ページ）で、信頼オブジェクトを操作して検証の動作を改変する、一般的な方法を説明します。その後、"信頼オブジェクトとNSURLConnection"（30 ページ）および"信頼オブジェクトとNSStream"（32 ページ）で、個々のネットワーク技術に改変した検証動作を組み込む方法を示します。

信頼オブジェクトを操作する

信頼オブジェクトを操作する細かい手順は、改変しようとする箇所に依存します。よく改変の対象となるのは、ホスト名とアンカ集合の2つです。ホスト名は、連鎖の末端に位置する認証局が発行した証明書の共通名、またはSAN（Subject Alternate Name）拡張に現れるいずれかの名前と一致しなければなりません。また、アンカ集合は、信頼の連鎖をたどって最終的に到達するべき、既知の（信頼できる）認証局を表します。

信頼できるアンカ証明書のリストに新しい証明書を追加する処理は、既存のアンカ証明書を配列にコピーし、これを可変配列にしたものを生成し、この配列に新しいアンカ証明書を追加し、今後はこの配列を使うよう信頼オブジェクトに指示する、という手順になります。これを行う簡単な関数例をリスト 1に示します。

リスト 1　アンカ証明書を`SecTrustRef`オブジェクトに追加するコード例

```
SecTrustRef addAnchorToTrust(SecTrustRef trust, SecCertificateRef trustedCert)
{
#ifdef PRE_10_6_COMPAT
        CFArrayRef oldAnchorArray = NULL;

        /* OS X 10.6より古い版であれば、組み込みの
           アンカを新しい配列にコピーする。*/
        if (SecTrustCopyAnchorCertificates(&oldAnchorArray) != errSecSuccess) {
                /* 何かがうまくいっていない。*/
                return NULL;
        }

        CFMutableArrayRef newAnchorArray = CFArrayCreateMutableCopy(
                kCFAllocatorDefault, 0, oldAnchorArray);
        CFRelease(oldAnchorArray);
#else
        /* iOSおよびOS X v10.6以降であれば、空の配列を
           生成するだけでよい。*/
        CFMutableArrayRef newAnchorArray = CFArrayCreateMutable (
                kCFAllocatorDefault, 0, &kCFTypeArrayCallBacks);
#endif

        CFArrayAppendValue(newAnchorArray, trustedCert);
```

```
        SecTrustSetAnchorCertificates(trust, newAnchorArray);

#ifndef PRE_10_6_COMPAT
        /* iOSおよびOS X v10.6以降であれば、独自のアンカを追加した後、
           組み込みのアンカを改めて有効にする。
         */
        SecTrustSetAnchorCertificatesOnly(trust, false);
#endif

        return trust;
```

注意: コード例のtrustedCert（SecCertificateRefオブジェクト）は、DER（Distinguished Encoding Rules）でエンコードされた証明書をCFDataオブジェクトにロードし、SecCertificateCreateWithDataを実行することにより生成します。

他のエンコーディングの証明書をロードするなど、より細かく制御したい場合は、SecItemImport関数も使ってください（OS X v10.7以降）。

ホスト名を改変する（あるサイト用の証明書が別のサイトでも使えるようにする、あるいはIPアドレスを指定して接続した場合にも使えるようにする）ためには、信頼ポリシーが証明書の解釈方法を判断するために使う、ポリシーオブジェクトを置き換える必要があります。そのためにはまず、当該ホスト名を設定した、TLSポリシーオブジェクトを生成してください。次に配列を生成してこのポリシーを収容します。最後に、信頼オブジェクトに対し、今後この配列を使うよう指示します。リスト 2 に、その処理をする関数の実装例を示します。

リスト 2　SecTrustRefオブジェクトの遠隔ホスト名を変更するコード例

```
SecTrustRef changeHostForTrust(SecTrustRef trust)
{
        CFMutableArrayRef newTrustPolicies = CFArrayCreateMutable(
                kCFAllocatorDefault, 0, &kCFTypeArrayCallBacks);

       SecPolicyRef sslPolicy = SecPolicyCreateSSL(true, CFSTR("www.example.com"));

        CFArrayAppendValue(newTrustPolicies, sslPolicy);
```

```
#ifdef MAC_BACKWARDS_COMPATIBILITY
        /* この方法が使えるのはOS X (v10.5以降) のみ */

        SecTrustSetPolicies(trust, newTrustPolicies);
        CFRelease(oldTrustPolicies);

        return trust;
#else
        /* この方法が使えるのはiOS 2以降または
           OS X v10.7以降 */

        CFMutableArrayRef certificates = CFArrayCreateMutable(
                kCFAllocatorDefault, 0, &kCFTypeArrayCallBacks);

        /* 元の信頼オブジェクトから証明書をコピーする */
        CFIndex count = SecTrustGetCertificateCount(trust);
        CFIndex i=0;
        for (i = 0; i < count; i++) {
                SecCertificateRef item = SecTrustGetCertificateAtIndex(trust, i);
                CFArrayAppendValue(certificates, item);
        }

        /* 新しい信頼オブジェクトを生成する */
        SecTrustRef newtrust = NULL;
       if (SecTrustCreateWithCertificates(certificates, newTrustPolicies, &newtrust)
 != errSecSuccess) {
                /* おそらくログを出力する位置として最適。*/

                return NULL;
        }

        return newtrust;
#endif
}
```

信頼オブジェクトとNSURLConnection

信頼の連鎖に沿って証明書を検証する、というNSURLConnectionの動作を改変するためには、2つのメソッドをオーバーライドする必要があります。

- `connection:canAuthenticateAgainstProtectionSpace:`

  このメソッドはNSURLConnectionに対し、ある種類の認証手順を知っている（したがってアプリケーション側が実施する）旨を伝えます。サーバ証明書を信頼してよいかどうか、アプリケーションが判断する場合、これは「アプリケーションがサーバを認証する」という（特別な）認証方式になるのです。

- `connection:didReceiveAuthenticationChallenge:`

  このメソッドで、サーバまたはクライアントが指定した信頼ポリシー、キー、ホスト名を修正して、信頼ポリシーが正常に評価されるようにします。

リスト3に、上記のメソッドの実装例を示します。

リスト3　NSURLConnectionオブジェクトが使う信頼オブジェクトをオーバーライドする例

```
// OS X 10.7以降あるいはiOS 5以降を対象とする場合、
// 第1のメソッドは省き、第2のメソッドを
// connection:willSendRequestForAuthenticationChallenge:メソッドとして使う。
// 旧オペレーティングシステムの場合、第1のメソッドを有効にし、
// 第2のメソッドをconnection:didReceiveAuthenticationChallenge:メソッドとして使う。
//


#ifndef NEW_STYLE
- (BOOL)connection:(NSURLConnection *)connection
canAuthenticateAgainstProtectionSpace:(NSURLProtectionSpace *)protectionSpace {
    #pragma unused(connection)


    NSString *method = [protectionSpace authenticationMethod];
    if (method == NSURLAuthenticationMethodServerTrust) {
            return YES;
    }
```

```
    return NO;
}

-(void)connection:(NSURLConnection *)connection
       didReceiveAuthenticationChallenge:(NSURLAuthenticationChallenge *)challenge
#else
-(void)connection:(NSURLConnection *)connection
        willSendRequestForAuthenticationChallenge:(NSURLAuthenticationChallenge
*)challenge
#endif
{
    NSURLProtectionSpace *protectionSpace = [challenge protectionSpace];
    if ([protectionSpace authenticationMethod] ==
NSURLAuthenticationMethodServerTrust) {
    SecTrustRef trust = [protectionSpace serverTrust];

    /***** ここで信頼ポリシーに変更を施す。*****/

    /* 信頼ポリシーを評価し直す。*/
    SecTrustResultType secresult = kSecTrustResultInvalid;
    if (SecTrustEvaluate(trust, &secresult) != errSecSuccess) {
        /* 信頼性の評価に失敗した。*/

        [connection cancel];

        // 必要に応じてその他の後始末を行う。
        return;
    }

    switch (secresult) {
        case kSecTrustResultUnspecified: // OSはこの証明書を暗黙のうちに信頼する。
        case kSecTrustResultProceed: // ユーザがOSに対し、これは信頼してよいと明言した。
            {
            NSURLCredential *credential =
                [NSURLCredential
credentialForTrust:challenge.protectionSpace.serverTrust];
```

```
            [challenge.sender useCredential:credential
forAuthenticationChallenge:challenge];
            return;
            }
        default:
            /* 認識できないキーである。そのまま次に進む。*/
    }
    /* サーバは信頼できないキーを送信した。*/
    [connection cancel];

    // 必要に応じてその他の後始末を行う。
    }
}
```

信頼オブジェクトとNSStream

NSStreamの信頼オブジェクトを改変する方法は、その内容に依存します。

他のTLSホスト名を指定したいだけであれば、ストリームを開く前に、3行のコードを実行すればできてしまいます。

リスト 4　NSStreamでTLSホスト名を改変するコード例

```
        NSDictionary *sslSettings =
                [NSDictionary dictionaryWithObjectsAndKeys:
                        @"www.gatwood.net",
                        (__bridge id)kCFStreamSSLPeerName, nil];
        if (![myInputStream setProperty: sslSettings
            forKey: (__bridge NSString *)kCFStreamPropertySSLSettings]) {
                // エラー処理。
        }
```

この結果、ストリームが認識するホスト名が変わり、ストリームオブジェクトが信頼オブジェクトを生成する際にはこの名前を渡すようになります。

信頼できるアンカのリストを実際に変更する必要がある場合、処理はもう少し複雑になります。ストリームオブジェクトは、信頼オブジェクトの生成後、すぐにその評価を試みます。信頼の評価に失敗した場合、信頼オブジェクトを改変する前にストリームは閉じられてしまいます。したがって、この処理をオーバーライドするためには、次のようにしなければなりません。

- 信頼の連鎖に沿って行うTLS検証を無効にする。ストリームが信頼の評価を行わなければ、失敗することもないので、ストリームが閉じてしまうこともありません。
- ストリームのデリゲートで、（信頼オブジェクトを改変した後）独自の方法で検証を行う。

ストリームデリゲートのイベントハンドラが呼び出される（ソケットがデータを処理できる状態になる）までに、オペレーティングシステムはTLSチャンネルを構築し、接続の他端から証明書チェーンを取得し、これを評価する信頼オブジェクトを生成してしまっています。この時点でTLSストリームが開いていますが、他端のホストが信頼できるかどうかは不明です。信頼の連鎖に沿って行う検証を無効にしたので、他端のホストが信頼できるかどうか、*自分*で検証しなければなりません。したがって、次の方針に従う必要があります。

- 非TLSポリシーを生成する、ホスト名として`NULL`ポインタを渡すなどして、ホスト名の検査を無効にしないでください。意図的に、証明書に記載されていないホスト名で接続する場合、当該ホストから受け取った証明書が、*自分が制御する*他のドメインで有効である場合に限って、処理を許可するべきです。
- 自己署名証明書をアンカとして暗黙に信頼（`kSecTrustOptionImplicitAnchors`）してはいけません。代わりに、独自の（自己署名）認証局証明書を、信頼できるアンカのリストに追加してください。
- 他のセキュリティオプション（無効になった証明書やルートの検査など）を、自分の判断で無効にしないでください。これに意味がある状況もありますが（2001年に署名されたドキュメントが、2001年に有効であった証明書を使って署名されていることを検証する、など）、ネットワーク処理に関する限り、一般に、デフォルトオプションのままにしておくべきです。

以上の方針に基づき、リスト 5に、`NSStream`で独自のTLSアンカ証明書を使う方法を示します。この例では、関数`addAnchorToTrust`（リスト 1 （27 ページ）を参照）も使っています。

リスト 5　`NSStream`で独自のTLSアンカ証明書を使うコード例

```
/* ソケット生成後に実行するコード：*/

        [inStream setProperty:NSStreamSocketSecurityLevelNegotiatedSSL
            forKey:NSStreamSocketSecurityLevelKey];

    NSDictionary *sslSettings =
```

```
        [NSDictionary dictionaryWithObjectsAndKeys:
        (id)kCFBooleanFalse, (id)kCFStreamSSLValidatesCertificateChain,
        nil];

    [inStream setProperty: sslSettings forKey: (__bridge NSString
*)kCFStreamPropertySSLSettings];



...



/* ストリームのデリゲートクラスに実装するメソッド */

NSString *kAnchorAlreadyAdded = @"AnchorAlreadyAdded";

- (void)stream:(NSStream *)theStream handleEvent:(NSStreamEvent)streamEvent
{
    if (streamEvent == NSStreamEventHasBytesAvailable || streamEvent ==
NSStreamEventHasSpaceAvailable) {
        /* 検査する。*/

        NSArray *certs = [theStream propertyForKey: (__bridge NSString
*)kCFStreamPropertySSLPeerCertificates];
        SecTrustRef trust = (SecTrustRef)[theStream propertyForKey: (__bridge
NSString *)kCFStreamPropertySSLPeerTrust];

        /* 証明書の配列が大きくなるのは望ましくないので、
           アンカを信頼リストに追加するのは、
           （毎回ではなく）最初にデータを受信した時点で行う。
         */
        NSNumber *alreadyAdded = [theStream propertyForKey: kAnchorAlreadyAdded];
        if (!alreadyAdded || ![alreadyAdded boolValue]) {
            trust = addAnchorToTrust(trust, self.trustedCert); // 先に定義済み。
            [theStream setProperty: [NSNumber numberWithBool: YES] forKey:
kAnchorAlreadyAdded];
        }
```

```
        SecTrustResultType res = kSecTrustResultInvalid;

        if (SecTrustEvaluate(trust, &res)) {
            /* 何らかの理由で信頼性の評価に失敗した。
               これはおそらく、証明書が破損しているか、
               コードに何か誤りがあることを意味する。*/

            /* 入力ストリームを後始末する。*/
            [theStream removeFromRunLoop: ... forMode: ...];
            [theStream setDelegate: nil];
            [theStream close];

            /* 出力ストリームを後始末する。*/
            ...

            return;

        }

        if (res != kSecTrustResultProceed && res != kSecTrustResultUnspecified) {
            /* ホストは信頼できない。*/
            /* 入力ストリームを後始末する。*/
            [theStream removeFromRunLoop: ... forMode: ...];
            [theStream setDelegate: nil];
            [theStream close];

            /* 出力ストリームを後始末する。*/
            ...

        } else {
            // ホストは信頼できる。データコールバックを通常通り実行する。

        }
    }
}
```

書類の改訂履歴

この表は「*ネットワーキング プログラミング トピックス*」の改訂履歴です。

日付	メモ
2013-09-17	コード例に現れる、iOSでは使えないシンボルについて、代替コードを追加しました。
2012-12-13	getsocknameに長さを渡すためのデータ型を修正しました。コード例に「&」が欠落している箇所があったので修正しました。sockaddr_in6の初期化コードを修正しました。
2012-07-19	ネットワーク処理の概要を説明したドキュメントには掲載されなかったが、比較的よく現れる処理をいくつか取り上げて解説した新規ドキュメント。

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
U.S.A.

Apple Japan
〒106-6140 東京都港区六本木 6
丁目10番1号 六本木ヒルズ
http://www.apple.com/jp

Offline copy. Trademarks go here.