

ここでは、pingコマンドのソースコードを調べ、まとめた内容を記録していきます。

開始日	2007年01月23日
最終更新日	2009年06月01日

はじめに

目標

pingソースを読むにあたって、それぞれ次のような目標を進めます。

- kato「パケット送信～受信の流れを掴み、自分でもpingパケットの送受信ができるようになる。pingコマンドのオプションに詳しくなる」
- ichi「コマンドオプションの取り扱いやパケット送受信の処理方法などの定石を知り身につける。各ファイルの意味・役割についてもしっかり理解する。C言語を思い出す：P」

使用ソースコード

GNUソフトウェア[inetutils-1.5](#)を使用します。コンパイル・インストール方法は[ping \(メモ\)](#)を参照してください。

TODO

進めていく中で、やること・やりたいことを書いてください。終わったら黒く塗りつぶすこと。必要に応じて細分化してもオッケーです。

指定可能なオプションの一覧作成
各ファイルの用途をまとめる
オプション未指定で起動したときの処理フローを調べる。関数コールの流れが分かると良い
ping_echo.cの調査。他のソースはping_echo.cが終わってから
パケット送信処理のルールを追う。ping_echo.cでのパケット生成～送信の流れをまとめる
パケット受信処理のルールを追う。パケットの受信待ち～受信～情報取得～画面表示の流れをまとめる

がんばりましょう。

外部インタフェース

この章では外から見えるpingの振る舞いについて記述します。

そもそも、pingコマンドって何なの？

ICMP ECHO_REQUESTパケットをネットワーク上のホストに送信するプログラムです。対象ホストが到達可能かどうかをテストします。

pingプログラムはICMPのECHO_REQUESTメッセージを送信し、ホストやゲートウェイから返信されるICMPのECHO_REPLYメッセージを受信します。ECHO_REQUESTメッセージはIPとICMPヘッ

ダーを持っていて、送信した時刻(struct timeval)と指定したサイズぶんのパディングデータによって構成されています。*1

指定可能なオプション

とりあえずping.8から抜粋。後でソースと比較し、きちんとまとめます。

オプション	処理(ロングオプション)
-V	<--version> バージョン情報を表示して終わる
-L	<--license> ライセンス情報を表示して終わる
-h	<--help> 使用方法を表示して終わる
-c (count)	<--count> ECHO_REQUESTの最大送信パケット数(またはECHO_REPLYを受信した数)で送信を止める
-d	<--debug> ソケットにSO_DEBUGオプションを付けて実行する
-r	<--ignore-routing> 通常の経路テーブルを無視(バイパス)し、接続されたネットワークのホストにダイレクトに送信する。もし対象ホストが接続されたネットワークに存在しない場合はIraが返る。このオプションは(当該ホストへの)経路情報を持たないローカルホストに対して有効である。(e.g., after the interface was dropped by routed(8)).
-s (size)	<--size> ECHO_REQUESTパケットのサイズを指定する。デフォルトは56bytesで、これはICMPヘッダを含めるとちょうど64bytesになる。
-i (wait)	<--interval> 各ECHO_REQUESTの送信間隔を秒で指定する。デフォルトは1秒。このオプションは-fと同時に設定することはできない。
-n	<--numeric> IPアドレスを10進数表記により出力する。ホスト名をルックアップする処理は行わない。
-v	<--verbose> 標準出力に対し情報を出来る限り出力する。ECHO_REPLY以外のICMPパケットも受信すれば出力する。
-t (type)	<--type ("echo"/"timestamp"/"address"/"router")> ICMPのメッセージタイプを設定する。このオプションは次に示す3つのオプションのどれかを設定したときと同じ結果を生む
-	<--echo> ICMP echoメッセージを送信する(デフォルト動作)
-	<--timestamp> ICMP timestampメッセージを送信する
-	<--address> ICMP addressメッセージを送信する
-	<--router> ICMP router discoveryメッセージを送信する。現在は未対応

-f	<--flood> 100回/秒かそれ以上のパケットを送信する、「pingの洪水（flood）」を発生させる。ECHO_REQUESTを送信するごとにピリオド「.」を1文字出力し、ECHO_REPLYが届くごとに削除する。したがって画面上に表示されたピリオドの数が破棄・消失したpingパケットの数として確認できる。このオプションはネットワークに対して強烈な負荷をかけるため、スーパーユーザーでのみ設定可能
-l (preload)	<--preload> 通常のping送信処理に入る前に、preloadで指定した個数のECHO_REQUESTを出来るだけ早く送信する。preload={1,65535}
-p (pattern)	<--pattern> パディングする値を16進数で設定する。このオプションはデータが崩れる問題を抱えたネットワークで有効である。たとえば"-p ff"と設定した場合、ペイロード部のビットは1で埋められる。
-q	<--quiet> 開始 / 終了時のサマリー以外情報を出力しないようにする。
-R	<--route> pingパケットの通った経路を記録し表示する。ECHO_REQUESTにRECORD_ROUTEオプションをつけることで経路を記録していき、戻ってきたECHO_REPLYに含まれる情報を表示する。 注意：IPヘッダに記録しておける最大経路数(9?)には注意すること。(残念ながら)多くのホスト・ルータではこのオプションを無視する。

出力情報の見方

送信中

```
# ping hoge_host
PING hoge_host (192.168.0.3) 56 bytes of data.
64 bytes from 192.168.0.3: icmp_seq=1 ttl=64 time=0.114 ms
64 bytes from 192.168.0.3: icmp_seq=2 ttl=64 time=0.176 ms
64 bytes from 192.168.0.3: icmp_seq=3 ttl=64 time=0.170 ms
:
```

まず、先頭の出力：

```
PING hoge_host (192.168.0.3): 56 data bytes.
```

は、「hoge_host(IPアドレスは192.168.0.3)に対しデータ長56バイトのECHO_REQUESTを送信する」という意味です。データ長はICMPヘッダとIPヘッダを除いたプリロード部のサイズを表します。

次の出力；

```
64 bytes from 192.168.0.3: icmp_seq=2 ttl=64 time=0.176 ms
```

は、ECHO_REQUESTに対するECHO_REPLYパケットを受信する度に出力されていきます。

最初の「64bytes」は送信時と違いICMPヘッダ + データ部の合計を表示していることに注意してください。

「icmp_seq=X」はECHO_REQUESTに付けられたシーケンス番号であり、この値の連続性を見るこ

とで、どのパケットが破棄されたかが分かります。

「ttl」はECHO_REPLYのttlフィールドの値を表します。この値はIPルーティングされるたびにデクリメントされるので、ECHO_REPLYが幾つルータを経由してきたかが分かります。言い換えると、ECHO_REQUESTがルータを経由した回数は分からないことを表しています。

「time」はECHO_REQUESTを送信してからECHO_REPLYを受信するまでに要した往復時間(Round Trip Time)を表示します。厳密には、

Start	ECHO_REQUEST送信時にデータ部に付与しておいた時刻情報
End	情報を出力しようとしたときの現在時刻

を調べ、両者の差をRTTとして出力しています。なおWindowsのECHO_REQUESTパケットには時刻情報がセットされていないため、PC内部で時刻情報を管理・出力していると思われる。

一方、ECHO_REPLYがタイムアウト時間までに届かなかった場合は以下の情報が出力されます。

```
64 bytes from 192.168.0.1: Destination Host Unreachable(etc.)
Vr HL TOS Len ID Flg off TTL Pro cks Src Dst Data
4 5 00 5400 0000 0 0040 40 01 9c02 192.168.0.1 192.168.0.60
```

エラーの理由と、送信したIPパケットの情報を出力します。

この後Ctrl+c(SIGINTシグナル)を入力するか、または-cで指定した個数のpingを送信完了すると以下のサマリー情報を出力して終了します。

```
--- hoge_host ping statistics ---
10 packets transmitted, 10 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.120/0.259/0.659/0.231 ms
#
```

上記は

「10パケットを送信したうち10パケット受信し、破棄率は0%。RTTの最小/平均/最大/標準偏差はそれぞれ0.120/0.259/0.659/0.231msだった」ことを表しています。これまで送信したパケットについての統計情報です。

ファイル構成

各ファイルの概要

pingフォルダ内のMakefile.am(automakeで使用するMakefileの種、みたいなもの)から抽出。ファイルはまだ他にもあるっぽいので、随時追加。

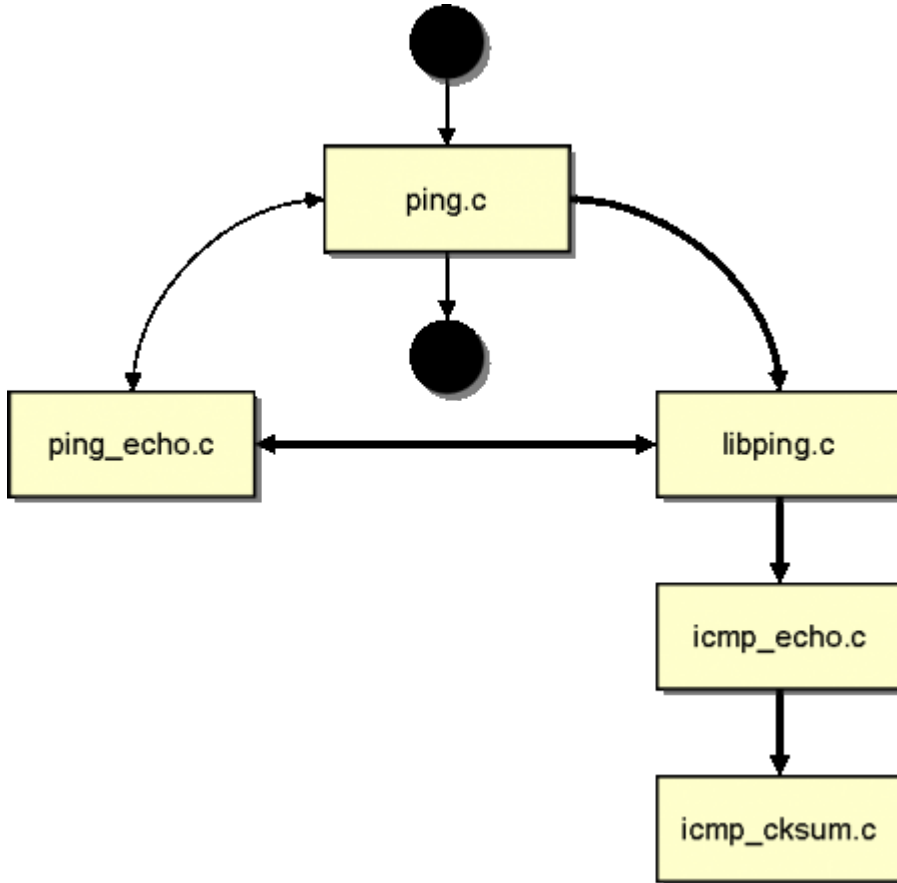
ファイル名	概要
ping.c	
ping_echo.c	
ping_address.c	
ping_router.c	
ping_timestamp.c	

ping_impl.h

ping構成ソース全般でインクルードするヘッダファイル。マクロ（用途不明）と時間の統計情報を計算するための構造体が定義されている。

各ファイルのつながり

「ping <IPアドレス> -c 10」と実行した時の、pingの各ファイル間の構成は次のとおりです。エントリーポイントはping.cです。



図の太い線は、Pingが送受信されるごとに関数がコールされているものを表しています。細い線は1回の実行において1回しか呼ばれません。

処理の流れ

基本的な処理の流れ（IPアドレスだけ指定したとき）

<関数がどのようにコールされていくかを簡単な説明付きで記述してください。オプション解析 ping開始準備 送信 受信 情報出力という流れを関数コールと紐付けられればベターです。>

関数のコールグラフ

「ping localhost -c 4」と実行したときの、関数のコールグラフは[これ](#)です。

このコールグラフはpvtraceで作成したトレース情報をgraphvizでグラフ化しています。グラフ作成の元になったdotファイルは、[ここ](#)からダウンロードできます。

モジュール構成図

コールグラフを元に、機能面で抽象化したときのモジュール構成図は以下のとおりです。

<モジュール構成を図示してください>

パケット送信

pingプログラムがパケットを送信する時、どのように送信しているかをまとめます。

- ソケットの操作
- パケットフィールドへのセット

パケット受信

pingプログラムがパケットを受信するとき、どのようにしているかをまとめます。

- ソケットの操作
- パケットフィールドからの情報取得

各関数の処理詳細

各関数の詳細は、長くなるので別ページ[ping \(内部関数説明 \)](#)に記載してあります。

ただしこの章の優先度は低いです。調べていく中で余裕があるときに書いてください。

使用しているテクニック・パターン

モジュール構成、デザインパターンや処理内容など、特筆すべき内容をここに書きます。

関数ポインタによるポリモーフィズム

pingコマンドで送信するICMPメッセージが異なると、ヘッダ情報、ペイロード、格納データなどがそれぞれで異なります。この処理を1処理の中でswitch文などにより場合分けをしてもよいのですが、pingでは関数ポインタによって処理を分割・抽象化しています。

ping.cのmain()関数でオプションを解析している中で、オプションによってはdecode_type()関数をコールします。この関数では、引数で渡された文字列を見て次のようにping_type変数に関数ポインタを代入しています。

```
if (strcasecmp (optarg, "echo") == 0)
    ping_type = ping_echo;
else if (strcasecmp (optarg, "timestamp") == 0)
    ping_type = ping_timestamp;
else if (strcasecmp (optarg, "address") == 0)
    ping_type = ping_address;
#if 0
else if (strcasecmp (optarg, "router") == 0)
    ping_type = ping_router;
#endif
```

各関数と送信するICMPメッセージの種類は以下のとおりです。

メッセージ	使用関数
ICMP echoメッセージ	ping_echo()
ICMP timestampメッセージ	ping_timestamp()
ICMP addressメッセージ	ping_address()
ICMP router discoveryメッセージ(未対応)	ping_router()

ping_type変数はping.c内でグローバル変数として定義されています（グローバル変数なのはmain()とdecode_type()の2箇所で利用されるからなのですが、その程度ならmain()のローカル変数で済むようにしたほうがスマートな気がします）。

```
int (*ping_type) (int argc, char **argv) = ping_echo;
```

int型のargc,char**型のargvの2引数を持つ関数へのポインタ（初期値はping_echo）、という意味です。

そして、main()ではオプションの解析が終了したあと最後に

```
return (*ping_type)(argc, argv);
```

とping_typeに格納された関数をコールして終わっています。ICMPメッセージの種別でコールする関数を分けたりしていません。

利点は、呼ばれた側では自分自身が行うべき処理のみを記述すれば良いためメンテナンス性や（関係ない他処理による）バグの混入を防げることです。もしこれを関数ポインタを使わないとしたら、冒頭に挙げた「ヘッダ情報、ペイロード、格納データ」それぞれについて場合分けをして設定処理を記述しなければいけませんので。

また、呼ぶ側の処理内容を統一できている点も利点の1つです。将来の処理追加に対して容易に対応できることが分かると思います。

static関数を別ファイルの関数からコールする

static関数であったとしても、別ファイルの関数からコールすることが出来ます。

関数のコールグラフを見ると、ping_echo.cの関数echo_finish()はping.cの関数ping_run()からコールされています。しかし、echo_finish()の関数定義を見ると、

```
static int echo_finish (void);
```

というようにstatic宣言されています。staticを関数に適用した場合、宣言されたファイルの外側からは見えなくなるのですが、なぜか別ファイルからコールできています。

これは、ping_run()に対してecho_finish()の関数ポインタを渡しているためです。

```
ping_echo (int argc, char **argv)
{
    :
    return ping_run (ping, echo_finish);
}

int ping_run (PING *ping, int (*finish)())
{
    :
    if (finish)
        return (*finish)();
}
```

ping_echo()からping_run()呼ぶときに、echo_finishのアドレスが渡されています。これによりping_run()でもecho_finish()を呼ぶときはどのアドレスにジャンプすればよいか分かるようになります（もちろん、ping_run()からecho_finish()を明示的にコールすることは出来ません）。

このように、関数ポインタを渡すことでstatic関数でも外部ソースファイルから参照・コールすることができます。関数を公開する範囲を限定することができる点が利点です。

後述するコールバック関数による疎結合の実現でも重要なテクニックです。

クロージャによる関数とデータの一括

まずここでのクロージャは、「関数コールに常に伴うデータ構造（状態保持可）」という意味で考えてください。クロージャ作成による外部データの束縛、というような意味ではないです。ping_echo.cの関数ping_echo()で、次のように関数を呼んでいます。

```
ping_set_event_handler (ping, handler, &ping_stat);
```

この最後の引数ping_statが、libping.cでは(void*)closureとして扱われ、この後の処理には必ず付いて回るようになっています。つまり、データ構造が処理に束縛されているのと同じ状況になっているのです。そして呼ばれる先々でデータ構造は更新されていき、分散や平均を求めるのに役立っているわけです。

コールバック関数による管理部と処理の分離

pingのソースは、

1. pingでのプロトコル動作を実現 ping.c
2. pingパケットの情報を表示 ping_echo.c
3. pingパケットのヘッダ部作成や実際の送受信 libping.c

というように、処理によってファイルが明確に分かれています。1はパケットを受信したときに何をするか（ここでは情報の表示）をするのみになっていて、スタート時にはこれら処理の関数ポインタをping.cからping_echo.cに登録し、その次はping_echo.cからlibping.cに処理を登録しています。関数を直接呼ばず、下位層にイベントハンドラとして登録することで、上位層に変更があっても下位層には影響を受けない、という関係になります。依存方向が片方向に限定されるわけです。粗結合なので、オプション次第で別のping_address.cやping_router.cに特定層を切り替えても問題はないわけです。

構造化プログラミングやオブジェクト指向プログラミングでも、こうした考えは重要ですので、ぜひマスターしたい知識です。

気づいた・面白い・分からない点

コードを読んでいく中で、気付いた点を以下に記します。

kato

- (2007/1/23)コマンド追加時にもできるだけ既存処理の修正が少なくなるように設計されているなあ。関数ポインタによる抽象化、コールバック関数によるイベントドリブン（もどき）、などなど。クロージャもある！いやいや、思っていたよりも構造化されていて、参考になるよコレ。思っていたよりも難しい、とも言えるわけだけど。
- (2007/2/7)main()でコマンドオプションを解析するときに、struct option long_options[]に対して長いオプションのvalメンバを1文字のオプション文字で設定している。そうか、こうすることで長いオプションと（同じ機能を持つ）短いオプションとを同じcase文の中で扱えるのか。細かいけれど、面白いテクニックだ。
- (2007/8/2)このプログラムでは、自動変数をヒープ変数のように使っていることに気がついた。ある関数から次の関数への自動変数の受け渡しがreturn文中での関数呼び出しと共に行われているせいで（末尾再帰チック）、読んだ先の処理が終わるまではスコープの中のままになり、逆にスコープを意識しないでも良いような状況になっている。しかも自動変数なので処理が終わ

れば解放をしてくれて・・・。
テクニックの1つだけど、これも面白い。
クラスやクロージャ、RAIIなどを言語そのものが提供していなくても、同じ機能を実装するテクニックは、機能を理解する上では大事なことだな、と思ったり。

ichi

- <気づいた点を書いてください。箇条書きにこだわる必要はないです>

コメント

他の人が書いた文章はいきなり修正せず、コメントやメールで一度連絡を取り合ってからにしましょう。あくまで仲良く。

- pingはBIOS画面の時点でレスポンスって返ってくるものなのでしょうか？
初歩的な質問になってしまうのですが
宜しくお願いします。 -- アナスタシア (2007-07-27 08:37:25)
- 普通は返ってこないと思います。

pingはカーネルの上で動く1つのサービスであって、ネットワークスタックが、もしくは少なくともEthernetデバイスドライバは動いていないといけません。
しかしBIOSの段階ではとても基本的なことしかできず (<http://community.osdev.info/?> (AT)BIOS参照)、
Ethernetデバイスが動いて、IPが動いて、というのは非現実的、というか無理だと思います。

なので、BIOSでPingが返ってくるようなものは無いと思います。
(「思います」口調なのは、WakeOnLANとかはどうなっているの？と言われるとよく分かってなくて・・・
すみません)

ちなみに逆に質問なのですが、どうしてそのような質問をされたんですか？
面白い(興味深い)質問でしたので、ぜひ聞かせてください。 -- kato (2007-07-29 23:55:27)