                  PPP Predictor Compression Protocol

Status of This Memo

   This memo provides information for the Internet community. This memo
   does not specify an Internet standard of any kind. Distribution of
   this memo is unlimited.

Abstract

   The Point-to-Point Protocol (PPP) [1] provides a standard method of
   encapsulating multiple protocol datagrams over point-to-point links.

   The PPP Compression Control Protocol [2] provides a method for
   transporting multi-protocol datagrams over PPP encapsulated links.

   This document describes the use of the Predictor data compression
   algorithm for compressing PPP encapsulated packets.

Table of Contents

1. Introduction

   Predictor is a high speed compression algorithm, available without
   license fees. The compression ratio obtained using predictor is not
   as good as other compression algorithms, but it remains one of the
   fastest algorithms available.

   Note that although care has been taken to ensure that the following
   code does not infringe any patents, there is no assurance that it is

Rand   Informational   [Page 1]

      not covered by a patent.

2. Licensing

      There are no license fees or costs associated with using the
      Predictor algorithm.

      Use the following code at your own risk.

3. Predictor Packets

      Before any Predictor packets may be communicated, PPP must reach the
      Network-Layer Protocol phase, and the Compression Control Protocol
      must reach the Opened state.

      Exactly one Predictor datagram is encapsulated in the PPP Information
      field, where the PPP Protocol field indicates type hex 00FD
      (compressed datagram).

      The maximum length of the Predictor datagram transmitted over a PPP
      link is the same as the maximum length of the Information field of a
      PPP encapsulated packet.

      Prior to compression, the uncompressed data begins with the PPP
      Protocol number. This value MAY be compressed when Protocol-Field-
      Compression is negotiated.

      PPP Link Control Protocol packets MUST NOT be send within compressed
      data.

3.1. Predictor theory

      Predictor works by filling a guess table with values, based on the
      hash of the previous characters seen. Since we are either emitting
      the source data, or depending on the guess table, we add a flag bit
      for every byte of input, telling the decompressor if it should
      retrieve the byte from the compressed data stream, or the guess
      table. Blocking the input into groups of 8 characters means that we
      don't have to bit-insert the compressed output - a flag byte preceeds
      every 8 bytes of compressed data. Each bit of the flag byte
      corresponds to one byte of reconstructed data.

Take the source file:

000000   4141 4141 4141 410a   4141 4141 4141 410a   AAAAAAA.AAAAAAA.
000010   4141 4141 4141 410a   4141 4141 4141 410a   AAAAAAA.AAAAAAA.
000020   4142 4142 4142 410a   4241 4241 4241 420a   ABABABA.BABABAB.
000030   7878 7878 7878 780a   xxxxxxx.

Compressing the above data yields the following:

```
000000     6041 4141 4141 0a60  4141 4141 410a 6f41     `AAAAA.`AAAAA.oA
000010     0a6f 410a 4142 4142  4142 0a60 4241 4241     .oA.ABABAB.`BABA
000020     420a 6078 7878 7878  0a                      B.`xxxxx.
```

Reading the above data says:

```
flag = 0x60 - 2 bytes in this block were guessed correctly, 5 and 6.
     Reconstructed data is:    0 1 2 3 4 5 6 7
        File:                  A A A A A
        Guess table:                     A A
flag = 0x60 - 2 bytes in this block were guessed correctly, 5 and 6.
     Reconstructed data is:    0 1 2 3 4 5 6 7
        File:                  A A A A A
        Guess table:                     A A
flag = 0x6f - 6 bytes in this block were guessed correctly, 0-3, 5 and 6.
     Reconstructed data is:    0 1 2 3 4 5 6 7
        File:                          A
        Guess table:           A A A A   A A
flag = 0x6f - 6 bytes in this block were guessed correctly, 0-3, 5 and 6.
     Reconstructed data is:    0 1 2 3 4 5 6 7
        File:                          A
        Guess table:           A A A A   A A
flag = 0x41 - 2 bytes in this block were guessed correctly, 0 and 6.
     Reconstructed data is:    0 1 2 3 4 5 6 7
        File:                    B A B A B
        Guess table:           A           A
flag = 0x60 - 2 bytes in this block were guessed correctly, 5 and 6.
     Reconstructed data is:    0 1 2 3 4 5 6 7
        File:                    B A B A B
        Guess table:                     A B
flag = 0x60 - 2 bytes in this block were guessed correctly, 5 and 6
     Reconstructed data is:    0 1 2 3 4 5 6 7
        File:                    x x x x x
        Guess table:                     x x
```

And now, on to the source - note that it has been modified to work
with a split block. If your data stream can't be split within a block
(e.g., compressing packets), then the code dealing with "final", and
the memcpy are not required. You can detect this situation (or
errors, for that matter) by observing that the flag byte indicates
that more data is required from the compressed data stream, but you
are out of data (len in decompress is <= 0). It *is* ok if len == 0,
and flags indicate guess table usage.

```
#include <stdio.h>
#ifdef __STDC__
```

```
    #include <stdlib.h>
    #endif
    #include <string.h>
    /*
     * pred.c -- Test program for Dave Rand's rendition of the
     * predictor algorithm
     * Updated by: iand@labtam.labtam.oz.au (Ian Donaldson)
     * Updated by: Carsten Bormann <cabo@cs.tu-berlin.de>
     * Original  : Dave Rand <dlr@bungi.com>/<dave_rand@novell.com>
     */

    /* The following hash code is the heart of the algorithm:
     * It builds a sliding hash sum of the previous 3-and-a-bit
     * characters which will be used to index the guess table.
     * A better hash function would result in additional compression,
     * at the expense of time.
     */
    #define HASH(x) Hash = (Hash << 4) ^ (x)

    static unsigned short int Hash;
    static unsigned char GuessTable[65536];

    static int
    compress(source, dest, len)
    unsigned char *source, *dest;
    int len;
    {
        int i, bitmask;
        unsigned char *flagdest, flags, *orgdest;

        orgdest = dest;
        while (len) {
            flagdest = dest++; flags = 0; /* All guess wrong initially */
            for (bitmask=1, i=0; i < 8 && len; i++, bitmask <<= 1) {
                if (GuessTable[Hash] == *source) {
                    flags |= bitmask; /* Guess was right - don't output */
                } else {
                    GuessTable[Hash] = *source;
                    *dest++ = *source; /* Guess wrong, output char */
                }
                HASH(*source++);len--;
            }
            *flagdest = flags;
        }
        return(dest - orgdest);
    }

    static int
```

```
    decompress(source, dest, lenp, final)
    unsigned char *source, *dest;
    int *lenp, final;
    {
        int i, bitmask;
        unsigned char flags, *orgdest;
        int len = *lenp;
        orgdest = dest;
        while (len >= 9) {
            flags = *source++;
            for (i=0, bitmask = 1; i < 8; i++, bitmask <<= 1) {
                if (flags & bitmask) {
                    *dest = GuessTable[Hash];       /* Guess correct */
                } else {
                    GuessTable[Hash] = *source;     /* Guess wrong */
                    *dest = *source++;          /* Read from source */
                    len--;
                }
                HASH(*dest++);
            }
            len--;
        }
        while (final && len) {
            flags = *source++;
            len--;
            for (i=0, bitmask = 1; i < 8; i++, bitmask <<= 1) {
                if (flags & bitmask) {
                    *dest = GuessTable[Hash];       /* Guess correct */
                } else {
                    if (!len)
                        break;  /* we seem to be really done -- cabo */
                    GuessTable[Hash] = *source;     /* Guess wrong */
                    *dest = *source++;          /* Read from source */
                    len--;
                }
                HASH(*dest++);
            }
        }
        *lenp = len;
        return(dest - orgdest);
    }

    #define SIZ1 8192

    static void
    compress_file(f) FILE *f; {
        char bufp[SIZ1];
        char bufc[SIZ1/8*9+9];
```

```
        int len1, len2;
        while ((len1 = fread(bufp, 1, SIZ1, f)) > 0) {
            len2 = compress((unsigned char *)bufp,
            (unsigned char *)bufc, len1);
            (void) fwrite(bufc, 1, len2, stdout);
        }
    }

    static void
    decompress_file(f) FILE *f; {
        char bufp[SIZ1+9];
        char bufc[SIZ1*9+9];
        int len1, len2, len3;

        len1 = 0;
        while ((len3 = fread(bufp+len1, 1, SIZ1, f)) > 0) {
            len1 += len3;
            len3 = len1;
            len2 = decompress((unsigned char *)bufp,
            (unsigned char *)bufc, &len1, 0);
            (void) fwrite(bufc, 1, len2, stdout);
            (void) memcpy(bufp, bufp+len3-len1, len1);
        }
        len2 = decompress((unsigned char *)bufp,
        (unsigned char *)bufc, &len1, 1);
        (void) fwrite(bufc, 1, len2, stdout);
    }

    int
    main(ac, av)
        int ac;
        char** av;
    {
        char **p = av+1;
        int dflag = 0;

        for (; --ac > 0; p++) {
            if (!strcmp(*p, "-d"))
                dflag = 1;
            else if (!strcmp(*p, "-"))
                (dflag?decompress_file:compress_file)(stdin);
            else {
                FILE *f = fopen(*p, "r");
                if (!f) {
                    perror(*p);
                    exit(1);
                }
                (dflag?decompress_file:compress_file)(f);
```
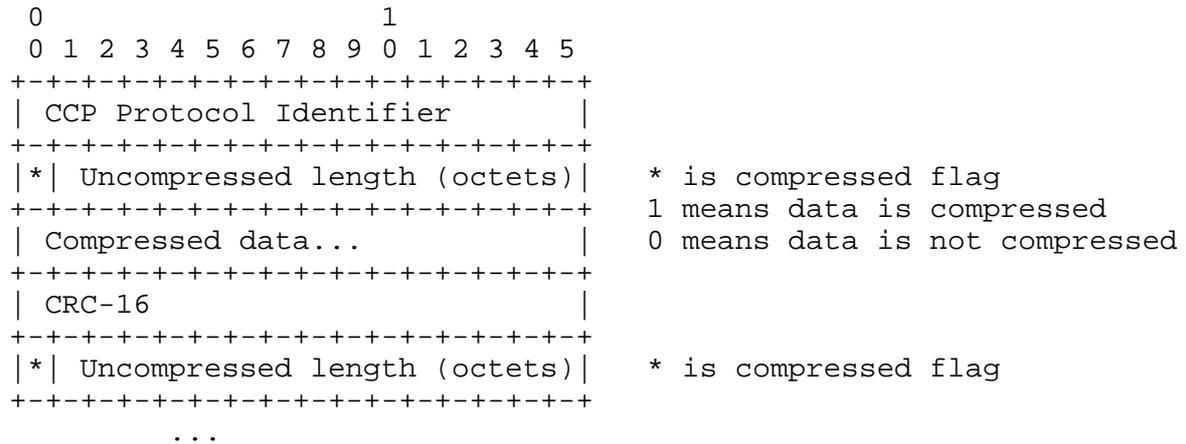
```
            (void) fclose(f);
          }
      }
      return(0);
   }
```

3.2. Encapsulation for Predictor type 1

   The correct encapsulation for type 1 compression is the protocol
   type, 1 bit indicating if the data is compressed or not, 15 bits of
   the uncompressed data length in octets, compressed data, and
   uncompressed CRC-16 of the two octets of unsigned length in network
   byte order, followed by the original, uncompressed data packet.

```
    0                   1
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   | CCP Protocol Identifier       |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |*| Uncompressed length (octets)|   * is compressed flag
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+   1 means data is compressed
   | Compressed data...            |   0 means data is not compressed
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   | CRC - 16                      |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

   The CCP Protocol Identifier that starts the packet is always 0xfd.
   If PPP Protocol field compression has not be negotiated, it MUST be a
   16-bit field.

   The Compressed data is the Protocol Identifier and the Info fields of
   the original PPP packet described in [1], but not the Address,
   Control, FCS, or Flag. The CCP Protocol field MAY be compressed as
   described in [1], regardless of whether the Protocol field of the CCP
   Protocol Identifier is compressed or whether PPP Protocol field
   compression has been negotiated.

   It is not required that any of the fields land on an even word
   boundary - the compressed data may be of any length. If during the
   decode procedure, the CRC-16 does not match the decoded frame, it
   means that the compress or decompress process has become
   desyncronized. This will happen as a result of a frame being lost in
   transit if LAPB is not used. In this case, a new configure-request
   must be sent, and the CCP will drop out of the open state. Upon
   receipt of the configure-ack, the predictor tables are cleared to
   zero, and compression can be resumed without data loss.

3.3. Encapsulation for Predictor type 2

   The correct encapsulation for type 2 compression is the protocol
   type, followed by the data stream. Within the data stream is the
   current frame length (uncompressed), compressed data, and
   uncompressed CRC-16 of the two octets of unsigned length in network
   byte order, followed by the original, uncompressed data. The data
   stream may be broken at any convenient place for encapsulation
   purposes. With type 2 encapsulation, LAPB is almost essential for
   correct delivery.

```
    0                   1
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   | CCP Protocol Identifier       |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |*| Uncompressed length (octets)|   * is compressed flag
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+   1 means data is compressed
   | Compressed data...            |   0 means data is not compressed
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   | CRC-16                        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |*| Uncompressed length (octets)|   * is compressed flag
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
           ...
```

   The CCP Protocol Identifier that starts the packet is always 0xfd.
   If PPP Protocol field compression has not be negotiated, it MUST be a
   16-bit field.

   The Compressed data is the Protocol Identifier and the Info fields of
   the original PPP packet described in [1], but not the Address,
   Control, FCS, or Flag. The CCP Protocol field MAY be compressed as
   described in [1], regardless of whether the Protocol field of the CCP
   Protocol Identifier is compressed or whether PPP Protocol field
   compression

   It is not required that any field land on an even word boundary - the
   compressed data may be of any length. If during the decode
   procedure, the CRC-16 does not match the decoded frame, it means that
   the compress or decompress process has become desyncronized. This
   will happen as a result of a frame being lost in transit if LAPB is
   not used. In this case, a new configure-request must be sent, and
   the CCP will drop out of the open state. Upon receipt of the
   configure-ack, the predictor tables are cleared to zero, and
   compression can be resumed without data loss.

4. Configuration Option Format

     There are no options for Predictor type one or two.

Security Considerations

     Security issues are not discussed in this memo.

References

     [1]   Simpson, W., "The Point-to-Point Protocol", STD 51, RFC
            1661, July 1994.

     [2]   Rand, D., "The PPP Compression Control Protocol (CCP)",
            RFC 1962, June 1996.

     [3]   Rand, D., "PPP Reliable Transmission", RFC 1663,
            July 1994.

Acknowledgments

     The predictor algorithm was originally implemented by Timo Raita, at
     the ACM SIG Conference, New Orleans, 1987.

     Bill Simpson helped with the document formatting.

Chair's Address

     The working group can be contacted via the current chair:

     Karl Fox
     Ascend Communications
     3518 Riverside Drive, Suite 101
     Columbus, Ohio 43221

     EMail: karl@ascend.com

Author's Address

     Questions about this memo can also be directed to:

     Dave Rand
     Novell, Inc.
     2180 Fortune Drive
     San Jose, CA 95131

     +1 408 321-1259
     EMail: dave_rand@novell.com