Using Self-Delimiting Numeric Values in Protocols
draft-eddy-dtn-sdnv-03

Status of this Memo

By submitting this Internet-Draft, each author represents that any
applicable patent or other IPR claims of which he or she is aware
have been or will be disclosed, and any of which he or she becomes
aware will be disclosed, in accordance with Section 6 of BCP 79.

Internet-Drafts are working documents of the Internet Engineering
Task Force (IETF), its areas, and its working groups. Note that
other groups may also distribute working documents as Internet-
Drafts.

Internet-Drafts are draft documents valid for a maximum of six months
and may be updated, replaced, or obsoleted by other documents at any
time. It is inappropriate to use Internet-Drafts as reference
material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at
http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at
http://www.ietf.org/shadow.html.

This Internet-Draft will expire on January 7, 2008.

Copyright Notice

Abstract

   Self-Delimiting Numeric Values (SDNVs) have recently been introduced
   as a field type within proposed Delay-Tolerant Networking protocols.
   The basic goal of an SDNV is to hold a non-negative integer value of
   arbitrary magnitude, without consuming much more space than
   necessary. The primary motivation is to conserve the bits sent
   across low-capacity or energy-intensive links typical of NASA deep-
   space missions, with a secondary goal of allowing the protocol to
   automatically adjust to unforseen usage scenarios. This can be
   desirable in that it allows protocol designers to avoid making
   difficult and potentially erroneous engineering decisions that may
   have to be hacked around in the future. This document describes
   formats and algorithms for SDNV encoding and decoding, and discusses
   implementation and usage of SDNVs.

Table of Contents

1. Introduction

This section begins by describing a common problem encountered in
network protocol engineering. It then provides some background on
the Self-Delimiting Numeric Values (SDNVs) proposed for use in Delay-
Tolerant Networking (DTN) protocols, and motivates their potential
applicability in other networking protocols.

1.1. Problems with Fixed Value Fields

Protocol designers commonly face an optimization problem in
determining the proper size for header fields. There is a strong
desire to keep fields as small as possible, in order to reduce the
protocol's overhead on the wire, and also allow for fast processing.
Since protocols can be used many years (even decades) after they are
designed, and networking technology has tended to change rapidly, it
is not uncommon for the use, deployment, or performance of a
particular protocol to be limited or infringed upon by the length of
some header field being too short. Two well-known examples of this
phenomenon are the TCP advertised receive window, and the IPv4
address length.

TCP segments contain an advertised receive window field that is fixed
at 16 bits [RFC0793], encoding a maximum value of around 65
kilobytes. The purpose of this value is to provide flow control, by
allowing a receiver to specify how many sent bytes its peer can have
outstanding (unacknowledged) at any time, thus allowing the receiver
to limit its buffer size. As network speeds have grown by several
orders of magnitude since TCP's inception, the combination of the 65
kilobyte maximum advertised window and long round-trip times
prevented TCP senders from being able to acheive the high-rates that
the underlying network supported. This limitation was remedied
through the use of the Window Scale option [RFC1323], which provides
a multiplier for the advertised window field. However, the Window
Scale multiplier is fixed for the duration of the connection,
requires bi-directional support, and limits the precision of the
advertised receive window, so this is certainly a less-than-ideal
solution. Because of the field width limit in the original design
however, the Window Scale is necessary for TCP to reach high sending
rates.

An IPv4 address is fixed at 32 bits [RFC0791] (as a historical note,
earlier versions of the IP specification supported variable-length
addresses). Due to the way that subnetting and assignment of address
blocks was performed, the number of IPv4 addresses has been seen as a
limit to the growth of the Internet [Hain05]. Two divergent paths to
solve this problem have been the use of Network Address Translators
(NATs) and the development of IPv6. NATs have caused a number of

side-issues and problems [RFC2993], leading to increased complexity
and fragility, as well as forcing work-arounds to be engineered for
many other protocols to function within a NATed environment. The
IPv6 solution's transitional work has been underway for several
years, but has still only begun to have visible impact on the global
Internet.

Of course, in both the case of the TCP receive window and IPv4
address length, the field size chosen by the designers seemed like a
good idea at the time. The fields were more than big enough for the
originally perceived usage of the protocols, and yet were small
enough to allow the total headers to remain compact and relatively
easy and efficient to parse on machines of the time. The fixed sizes
that were defined represented a tradeoff between the scalability of
the protocol versus the overhead and efficiency of processing. In
both cases, these engineering decisions turned out to be painfully
incorrect.

1.2. SDNVs for DTN Protocols

In proposals for the DTN Bundle Protocol (BP) [SB05] and Licklider
Transmission Protocol (LTP) [RBF06], SDNVs have been used for several
fields including identifiers, payload/header lengths, and serial
(sequence) numbers. SDNVs were developed for use in these types of
fields, to avoid sending more bytes than needed, as well as avoiding
fixed sizes that may not end up being appropriate. For example,
since LTP is intended primarily for use in long-delay interplanetary
communications [BRF06], where links may be fairly low in capacity, it
is desirable to avoid the header overhead of routinely sending a 64-
bit field where a 16-bit field would suffice. Since many of the
nodes implementing LTP are expected to be beyond the current range of
human spaceflight, upgrading their on-board LTP implementations to
use longer values if the defined fields are found to be too short
would also be problematic. Furthermore, extensions similar in
mechanism to TCP's Window Scale option are unsuitable for use in DTN
protocols since due to high delays, DTN protocols must avoid
handshaking and configuration parameter negotiation to the greatest
extent possible. All of these reasons make the choice of SDNVs for
use in DTN protocols particularly wise.

1.3. SDNV Usage

In short, an SDNV is simply a way of representing non-negative
integers (both positive integers of arbitrary magnitude and 0),
without expending too-much unneccessary space. This definition
allows SDNVs to represent many common protocol header fields, such
as:

o Random identification fields as used in the IPsec Security
   Parameters Index or in IP headers for fragment reassembly (Note:
   the 16-bit IP ID field for fragment reassembly was recently found
   to be too short in some environments [I-D.heffner-frag-harmful]),

o Sequence numbers as in TCP or SCTP,

o Values used in cryptographic algorithms such as RSA keys, Diffie-
   Hellman key-agreement, or coordinates of points on elliptic
   curves.

o Message lengths as used in file transfer protocols.

o Nonces and cookies.

o Etc.

The use of SDNVs rather than fixed length fields gives protocol
designers the ability to somewhat circumvent making difficult-to-
reverse field-sizing decisions, since the SDNV wire-format grows and
shrinks depending on the particular value encoded. SDNVs do not
necessarily provide optimal encodings for values of any particular
length, however they allow protocol designers to avoid potential
blunders in assigning fixed lengths, and remove the complexity
involved with either negotiating field lengths or constructing
protocol extensions.

To our knowledge, at this time, no protocols designed for use outside
of the DTN domain have proposed to use SDNVs, however there is no
inherent reason not to use SDNVs more broadly in the future. The two
examples cited here of fields that have proven too-small in general
Internet protocols are only a small sampling of the much larger set
of similar instances that the authors can think of.

Many protocols use a Type-Length-Value method for encoding variable
length strings (e.g. TCP's options format, or many of the fields in
IKEv2). An SDNV is equivalent to combining the length and value
portions of this type of field, with the overhead of the length
portion amortized out over the bytes of the value. The penalty paid
for this in an SDNV may be several extra bytes for long values (e.g.
1024 bit RSA keys). See Section 4 for further discussion and a
comparison.

As is shown in later sections, for large values, the current SDNV
scheme is fairly inefficient in terms of space (1/8 of the bits are
overhead) and not particularly easy to encode/decode in comparison to
alternatives. The best use of SDNVs may often be to define the
Length field of a TLV structure to be an SDNV whose value is the

length of the TLV's Value field. In this way, one can avoid forcing
large numbers from being directly encoded as an SDNV, yet retain the
extensibility that using SDNVs grants.

2. **Definition of SDNVs**

   An early definition of the SDNV format bore resemblance to the ASN.1
   [ASN1] Basic Encoding Rules (BER) [ASN1-BER] for lengths (Section
   8.1.3 of X.690). The current SDNV format is the one used by ASN.1
   BER for encoding tag identifiers greater than or equal to 31 (Section
   8.1.2.4.2 of X.690). A comparison between the current SDNV format
   and the early SDNV format is made in Section 4.

   The currently-used format is very simple. Before encoding, an
   integer is represented as a left-to-right bitstring beginning with
   its most significant bit, and ending with its least signifcant bit.
   On the wire, the bits are encoded into a series of bytes. The most
   significant bit of each wire format byte specifies whether it is the
   final byte of the encoded value (when it holds a 0), or not (when it
   holds a 1). The remaining 7 bits of each byte in the wire format are
   taken in-order from the integer's bitstring representation. If the
   bitstring's length is not a multiple of 7, then the string is left-
   padded with 0s.

   For example:

   o 1 (decimal) is represented by the bitstring "0000001" and encoded
      as the single byte 0x01 (in hexadecimal)

   o 128 is represented by the bitstring "10000001 00000000" and
      encoded as the bytes 0x81 followed by 0x00.

   o Other values can be found in the test vectors of the source code
      in Appendix A

   To be perfectly clear, and avoid potential interoperability issues
   (as have occurred with ASN.1 BER time values), we explicitly state
   two considerations regarding zero-padding. (1) When encoding SDNVs,
   any leading (most significant) zero bits in the input number might be
   discarded by the SDNV encoder. Protocols that use SDNVs should not
   rely on leading-zeros being retained after encoding and decoding
   operations. (2) When decoding SDNVs, the relevant number of leading
   zeros required to pad up to a machine word or other natural data unit
   might be added. These are put in the most-significant positions in
   order to not change the value of the number.

[3](#). **Basic Algorithms**

   This section describes some simple algorithms for creating and
   parsing SDNV fields. These may not be the most efficient algorithms
   possible, however, they are easy to read, understand, and implement.
   [Appendix A](#) contains Python source code implementing the routines
   described here. Only SDNV's of the currently-used form are
   considered in this section.

[3.1](#). **Encoding Algorithm**

   There is a very simple algorithm for the encoding operation that
   converts a non-negative integer (n, of length 1+floor(log_2 n) bits)
   into an SDNV. This algorithm takes n as its only argument and
   returns a string of bytes:

   o (Initial Step) Set the return value to a byte sharing the least
      significant 7 bits of n, and with 0 in the most significant bit,
      but do not return yet. Right shift n 7 bits and use this as the
      new n value. If implemented using call-by-reference rather than
      call-by-value, make a copy of n for local use at the start of the
      function call.

   o (Recursion Step) If n == 0, return. Otherwise, take the byte
      0x80, and bitwise-or it with the 7 least significant bits left in
      n. Set the return value to this result with the previous return
      string appended to it. Set n to itself shifted right 7 bits
      again. Repeat Recursion Step.

   This encoding algorithm can easily be seen to have time complexity of
   O(log_2 n), since it takes a number of steps equal to ceil(n/7), and
   no additional space beyond the size of the result (8/7 log_2 n) is
   required. One aspect of this algorithm is that it assumes strings
   can be efficiently appended to new bytes. One way to implement this
   is to allocate a buffer for the expected length of the result and
   fill that buffer one byte at a time from the right end.

[3.2](#). **Decoding Algorithm**

   Decoding SNDVs is a more difficult operation than encoding them, due
   to the fact that no bound on the resulting value is known until the
   SDNV is parsed, at which point the value itself is already known.
   This means that if space is allocated for decoding the value of an
   SDNV into, it is never known whether this space will be overflowed
   until it is 7 bits away from happening.

   (Initial Step) Set the result to 0. Set a pointer to the beginning
   of the SDNV.

(Recursion Step) Shift the result left 7 bits. Add the lower 7 bits
of the value at the pointer to the result. If the high-order bit
under the pointer is a 1, move the pointer right one byte and repeat
the Recursion Step, otherwise return the current value of the result.

This decoding algorithm takes no more additional space than what is
required for the result (7/8 the length of the SDNV) and the pointer.
The complication is that before the result can be left-shifted in the
Recursion Step, an implementation needs to first make sure that this
won't cause any bits to be lost, and re-allocate a larger piece of
memory for the result, if required. The pure time complexity is the
same as for the encoding algorithm given, but if re-allocation is
needed due to the inability to predict the size of the result, in
reality decoding may be slower.

4. **Comparison to Alternatives**

   This section compares three alternative ways of implementing the
   concept of SDNVs: (1) the TLV scheme commonly used in the Internet
   family, and many other families of protocols, (2) the old style of
   SDNVs (both the SDNV-8 and SDNV-16) defined in an early stage of
   LTP's development [BRF04], and (3) the current SDNV format.

   The TLV method uses two fixed-length fields to hold the Type" and
   Length elements that then imply the syntax and semantics of the
   "value" element. This is only similar to an SDNV in that the value
   element can grow or shrink within the bounds capable of being
   conveyed by the Length field. Two fundamental differences between
   TLVs and SDNVs are that through the Type element, TLVs also contain
   some notion of what their contents are semantically, while SDNVs are
   simply generic non-negative integers, and protocol engineers still
   have to pick fixed-lengths for the Type and Length fields in the TLV
   format.

   Some protocols use TLVs where the value conveyed within the Length
   field needs to be decoded into the actual length of the Value field.
   This may be accomplished through simple multiplication, left-
   shifting, or a look-up table. In any case, this tactic limits the
   granularity of the possible Value lengths, and can contribute some
   degree of bloat if Values do not fit neatly within the available
   decoded Lengths.

   In the SDNV format originally used by LTP, parsing the first byte of
   the SDNV told an implementation how much space was required to hold
   the contained value. There were two different types of SDNVs defined
   for different ranges of use. The SDNV-8 type could hold values up to
   127 in a single byte, while the SDNV-16 type could hold values up to
   32,767 in 2 bytes. Both formats could encode values requiring up to
   N bytes in N+2 bytes, where N<127. The two major difference between
   this old SDNV format and the currently-used SDNV format is that the
   new format is not as easily decoded as the old format was, but the
   new format also has absolutely no limitation on its length.

   The advantage in ease of parsing that the old format manifests itself
   in two aspects: (1) the size of the value is determinable ahead of
   time, in a way equivalent to parsing a TLV, and (2) the actual value
   is directly encoded and decoded, without shifting and masking bits as
   is required in the new format. For these reasons, the old format
   requires less computational overhead to deal with, but is also very
   limited, in that it can only hold a 1024-bit number, at maximum.
   Since according to IETF Best Current Practices, an asymmetric
   cryptography key needed to last for a long term requires using moduli
   of over 1228 bits [RFC3766], this could be seen as a severe

limitation of the old-style of SDNVs, which the currently-used style does not suffer from.

Table 1 compares the maximum values that can be encoded into SDNVs of various lengths using the old SDNV-8/16 method and the current SDNV method. The only place in this table where SDNV-16 is used rather than SDNV-8 is in the 2-byte row. Starting with a single byte, the two methods are equivalent, but when using 2 bytes, the old method is a more compact encoding by one-bit. From 3 to 7 bytes of length though, the current SDNV format is more compact, since it only requires one-bit per byte of overhead, whereas the old format used a full byte. Thus, at 8 bytes, both schemes are equivalent in efficiency since they both use 8 bits of overhead. Up to 129 bytes, the old format is more compact than the current one, although after this limit it becomes unusable.

Bytes	SDNV-8/16 Maximum Value	SDNV Maximum Value	SDNV-8/16 Overhead Bits	SDNV Overhead Bits
1	127	127	1	1
2	32,767	16,383	1	2
3	65,535	2,097,151	8	3
4	2^24 - 1	2^28 - 1	8	4
5	2^32 - 1	2^35 - 1	8	5
6	2^40 - 1	2^42 - 1	8	6
7	2^48 - 1	2^49 - 1	8	7
8	2^56 - 1	2^56 - 1	8	8
9	2^64 - 1	2^63 - 1	8	9
10	2^72 - 1	2^70 - 1	8	10
16	2^120 - 1	2^112 - 1	8	16
32	2^248 - 1	2^224 - 1	8	32
64	2^504 - 1	2^448 - 1	8	64
128	2^1016 - 1	2^896 - 1	8	128
129	2^1024 - 1	2^903 - 1	8	129
130	N/A	2^910 - 1	N/A	130
256	N/A	2^1792 - 1	N/A	256

Table 1

   In general, it seems like the most promising use of SDNVs may be to
   define the Length field of a TLV structure to be an SDNV whose value
   is the length of the TLV's Value field. This leverages the strengths
   of the SDNV format and limits the effects of its weaknesses.

5. **Security Considerations**

   The only security considerations with regards to SDNVs are that code
   which parses SDNVs should have bounds-checking logic and be capable
   of handling cases where an SDNV's value is beyond the code's ability
   to parse. These precautions can prevent potential exploits involving
   SDNV decoding routines.

   Stephen Farrell noted that very early definitions of SDNVs also
   allowed negative integers. This was considered a potential security
   hole, since it could expose implementations to underflow attacks
   during SDNV decoding. There is a precedent in that many existing TLV
   decoders map the Length field to a signed integer and are vulnerable
   in this way. An SDNV decoder should be based on unsigned types and
   not have this issue.

6. IANA Considerations

   This document has no IANA considerations.

7. **Acknowledgements**

8. Informative References

   [ASN1]   ITU-T Rec. X.680, "Abstract Syntax Notation One (ASN.1).
               Specification of Basic Notation", ISO/IEC 8824-1:2002,
               2002.

   [ASN1-BER]
               ITU-T Rec. X.690, "Abstract Syntax Notation One (ASN.1).
               Encoding Rules: Specification of Basic Encoding Rules
               (BER), Canonical Encoding Rules (CER) and Distinguished
               Encoding Rules (DER)", ISO/IEC 8825-1:2002, 2002.

   [BRF04]   Burleigh, S., Ramadas, M., and S. Farrell, "Licklider
               Transmission Protocol", draft-irtf-dtnrg-ltp-00 (expired),
               May 2004.

   [BRF06]   Burleigh, S., Ramadas, M., and S. Farrell, "Licklider
               Transmission Protocol - Motivation",
               draft-irtf-dtnrg-ltp-motivation-02 (work in progress),
               March 2006.

   [Hain05]   Hain, T., "A Pragmatic Report on IPv4 Address Space
               Consumption", Internet Protocol Journal Vol. 8, No. 3,
               September 2005.

   [I-D.heffner-frag-harmful]
               Heffner, J., "IPv4 Reassembly Errors at High Data Rates",
               draft-heffner-frag-harmful-05 (work in progress),
               May 2007.

   [RBF06]   Ramadas, M., Burleigh, S., and S. Farrell, "Licklider
               Transmission Protocol - Specification",
               draft-irtf-dtnrg-ltp-04 (work in progress), March 2006.

   [RFC0791]   Postel, J., "Internet Protocol", STD 5, RFC 791,
               September 1981.

   [RFC0793]   Postel, J., "Transmission Control Protocol", STD 7,
               RFC 793, September 1981.

   [RFC1323]   Jacobson, V., Braden, B., and D. Borman, "TCP Extensions
               for High Performance", RFC 1323, May 1992.

   [RFC2993]   Hain, T., "Architectural Implications of NAT", RFC 2993,
               November 2000.

   [RFC3766]   Orman, H. and P. Hoffman, "Determining Strengths For
               Public Keys Used For Exchanging Symmetric Keys", BCP 86,

                   RFC 3766, April 2004.

   [SB05]   Scott, K. and S. Burleigh, "Bundle Protocol
                   Specification", draft-irtf-dtnrg-bundle-spec-04 (work in
                   progress), November 2005.

Appendix A. SNDV Python Source Code

```
# sdnv_decode() takes a string argument s, which is assumed to be an
#   SDNV.  The function returns a pair of the non-negative integer n
#   that is the numeric value encoded in the SDNV, and and integer l
#   that is the distance parsed into the input string.  If the slen
#   argument is not given (or is not a non-zero number) then, s is
#   parsed up to the first byte whose high-order bit is 0 -- the
#   length of the SDNV portion of s does not have to be pre-computed
#   by calling code.  If the slen argument is given as a non-zero
#   value, then slen bytes of s are parsed.  The value for n of -1 is
#   returned for any type of parsing error.
#
# NOTE: In python, integers can be of arbitrary size.  In other
#   languages, such as C, SDNV-parsing routines should take
#   precautions to avoid overflow (e.g. by using the Gnu MP library,
#   or similar).
#
def sdnv_decode(s, slen=0):
  n = long(0)
  for i in range(0, len(s)):
    v = ord(s[i])
    n = n<<7
    n = n + (v & 0x7F)
    if v>>7 == 0:
      slen = i+1
      break
    elif i == len(s)-1 or (slen != 0 and i > slen):
      n = -1 # reached end of input without seeing end of SDNV
  return (n, slen)

# sdnv_encode() returns the SDNV-encoded string that represents n.
#   An empty string is returned if n is not a non-negative integer
def sdnv_encode(n):
  r = ""
  # validate input
  if n >= 0 and (type(n) in [type(int(1)), type(long(1))]):
    flag = 0
    done = False
    while not done:
      # encode lowest 7 bits from n
      newbits = n & 0x7F
      n = n>>7
      r = chr(newbits + flag) + r
      if flag == 0:
        flag = 0x80
      if n == 0:
        done = True
```

```
      return r


   # test cases from LTP and BP internet-drafts, only print failures
   def sdnv_test():
     tests = [(0xABC, chr(0x95) + chr(0x3C)),
              (0x1234, chr(0xA4) + chr (0x34)),
              (0x4234, chr(0x81) + chr(0x84) + chr(0x34)),
              (0x7F, chr(0x7F))]

     for tp in tests:
       # test encoding function
       if sdnv_encode(tp[0]) != tp[1]:
         print "sdnv_encode fails on input %s" % hex(tp[0])
       # test decoding function
       if sdnv_decode(tp[1])[0] != tp[0]:
         print "sdnv_decode fails on input %s, giving %s" % \
               (hex(tp[0]), sdnv_decode(tp[1]))
```

Author's Address

     Wesley M. Eddy
     Verizon Federal Network Systems
     NASA Glenn Research Center
     21000 Brookpark Rd, MS 54-5
     Cleveland, OH   44135

     Phone: 216-433-6682
     Email: weddy@grc.nasa.gov