CoRE Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 7, 2020

K. Hartke
Ericsson

November 4, 2019

Constrained Resource Identifiers
draft-ietf-core-href-01

Abstract

Constrained Resource Identifiers (CoRIs) are an alternate serialization of Uniform Resource Identifiers (URIs) that encodes the URI components in Concise Binary Object Representation (CBOR) instead of a string of characters. This simplifies parsing, reference resolution, and comparison of URIs in environments with severe limitations on processing power, code size, and memory size.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

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."

This Internet-Draft will expire on May 7, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 2
  1.1. Notational Conventions 3
2. Data Model 3
  2.1. Options 3
  2.2. Option Sequences 4
3. CBOR 7
4. Python 7
  4.1. Reference Resolution 9
  4.2. URI Recomposition 10
  4.3. CoAP Encoding 12
5. Security Considerations 14
6. IANA Considerations 14
7. References 14
  7.1. Normative References 14
  7.2. Informative References 15
Acknowledgements 15
Author's Address 15

1. Introduction

Uniform Resource Identifier (URI) references [RFC3986] are the standard way to link to resources in hypertext formats such as HTML [W3C.REC-html52-20171214] or the HTTP "Link" header field [RFC8288]. A URI reference is either a URI or a relative reference that must be resolved against a base URI.

URI references are strings of characters chosen from the repertoire of US-ASCII characters. The individual components of a URI reference are delimited by a number of reserved characters, which necessitates the use of percent-encoding when these reserved characters are used in a non-delimiting function. One component can also contain special dot-segments that affect how the component is to be interpreted. The resolution of URI references involves parsing the character string into its components, combining those components with the components of a base URI, merging path components, removing dot-segments, and recomposing the result back into a character string.

Overall, the proper processing of URIs is quite complicated. This can be a problem in particular in constrained environments [RFC7228], where devices often have severe code size limitations. As a result, many implementations in these environments choose to support only an ad-hoc, informally-specified, bug-ridden, non-interoperable subset of half of the URI standard.

This document introduces Constrained Resource Identifier (CoRI) references, an alternate serialization of URI references that encodes the URI components in Concise Binary Object Representation (CBOR) [RFC7049] instead of a string of characters. Assuming an implementation of CBOR is already present on a device, typical operations on URI references such as parsing, reference resolution, and comparison can be implemented more easily than for character strings. A full implementation that covers all corner cases is intended to be implementable in a relatively small amount of code.

As a result of the simplification, CoRI references are not capable of expressing all URI references permitted by the syntax of RFC 3986. (Hence the "constrained" in "Constrained Resource Identifiers".) The supported subset includes all Constrained Application Protocol (CoAP) URIs [RFC7252], most Hypertext Transfer Protocol (HTTP) URIs [RFC7230], and many other URIs that function as resource locators.

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Terms defined in this document appear in _cursive_ where they are introduced.

2. Data Model

The data model for CoRI references is very similar to the serialization of the request URI in CoAP messages [RFC7252]: The components of a URI reference are encoded as a sequence of _options_, where each path segment and query parameter becomes its own option. Every option consists of an _option number_ identifying the type of option (scheme, host name, path segment, etc.) and an _option value_.

2.1. Options

The following types of options are defined:

scheme
: Specifies the URI scheme. The option value can be any Unicode string matching the "scheme" rule described in Section 3.1 of RFC 3986 [RFC3986].

host.name

Specifies the host of the URI authority as a registered name. The option value can be any Unicode string matching the specifications of the URI scheme.

host.ip
Specifies the host of the URI authority as an IPv4 address or an IPv6 address. The option value is a byte string with a length of either 4 or 16 bytes, respectively.

port
Specifies the port number of the URI authority. The option value is an integer in the range from 0 to 65535.

path.type
Specifies the type of the URI path for reference resolution. The option value is an integer in the range from 0 to 127, named as follows:

```
0 - absolute-path
1 - append-relation
2 - append-path
3 - relative-path
4 - relative-path-1up
5 - relative-path-2up
6 - relative-path-3up
7 - relative-path-4up
...
```

path
Specifies one segment of the URI path. The option value can be any Unicode string with the exception of "." and "..". This option can occur more than once.

query
Specifies one argument of the URI query. The option value can be any Unicode string. This option can occur more than once.

fragment
Specifies the fragment identifier. The option value can be any Unicode string.

No percent-encoding is performed in option values.

2.2. Option Sequences

```
                _ host.name _
____ scheme __/             \___ port _
 \ \________/ \__ host.ip __/ /        \
  \___________________________/ ________/
   \                           / ________    _________
    \                         / /        \  /         \
     \___________ path.type __\_\_ path _/__\_ query _/__ fragment __
                 \____________/   \________/  \_________/ \__________/
```

Figure 1: Structure of a Well-Formed Sequence of Options

A sequence of options is considered _well-formed_ if:

- o the sequence of options is empty or starts with a "scheme", "host.name", "host.ip", "port", "path.type", "path", "query", or "fragment" option;

- o any "scheme" option is followed by either a "host.name" or a "host.ip" option;

- o any "host.name" option is followed by a "port" option;

- o any "host.ip" option is followed by a "port" option;

- o any "port" option is followed by a "path", "query", or "fragment" option or is at the end of the sequence;

- o any "path.type" option is followed by a "path", "query", or "fragment" option or is at the end of the sequence;

- o any "path" option is followed by a "path", "query", or "fragment" option or is at the end of the sequence;

- o any "query" option is followed by a "query" or "fragment" option or is at the end of the sequence; and

- o any "fragment" option is at the end of the sequence.

A well-formed sequence of options is considered _absolute_ if the sequence of options starts with a "scheme" option.

A well-formed sequence of options is considered _relative_ if the sequence of options is empty or starts with an option other than a "scheme" option.

An absolute sequence of options is considered _normalized_ if the result of resolving the sequence of options against any base is equal

to the input. (It doesn't matter what base it is resolved against, since it is already absolute.)

The following operations can be performed on a sequence of options:

resolve(href, base)
: Resolves a well-formed sequence of options `href` against an absolute sequence of options `base`. This operation MUST be performed by applying any algorithm that is functionally equivalent to the reference implementation in Section 4.1 of this document.

relative(href, base)
: Makes an absolute sequence of options `href` relative to an absolute sequence of options `base`. This operation MUST be performed by applying any algorithm that returns a sequence of options such that `resolve(relative(h, b), b)` is equal to `h` given the same `b`.

recompose(href)
: Recomposes a URI from an absolute sequence of options `href`. This operation MUST be performed by applying any algorithm that is functionally equivalent to the reference implementation in Section 4.2 of this document.

  To reduce variability, it is RECOMMENDED to uppercase the letters in the hexadecimal notation when percent-encoding octets [RFC3986] and to follow the recommendations of Section 4 of RFC 5952 for the text representation of IPv6 addresses [RFC5952].

decompose(str)
: Decomposes a URI `str` into a sequence of options. This operation MUST be performed by applying any algorithm that returns a sequence of options such that `recompose(decompose(x))` is equivalent to `x`.

coap(href)
: Constructs CoAP options from an absolute, normalized sequence of options. This operation MUST be performed by recomposing the sequence of options to a URI (as described above) and decomposing the URI into CoAP options (as specified in Section 6.4 of RFC 7252). A concise implementation of this algorithm is illustrated in Section 4.3 of this document.

3. CBOR

In Concise Binary Object Representation (CBOR) [RFC7049], a sequence of options is encoded as an array that contains the option numbers and option values in alternating order.

The structure can be described in the Concise Data Definition Language (CDDL) [RFC8610] as follows:

```
CoRI = [?(scheme:    1, text .regexp "[A-Za-z][A-Za-z0-9+.-]*"),
        ?(host.name: 2, text //
          host.ip:   3, bytes .size 4 / bytes .size 16),
        ?(port:      4, 0..65535),
        ?(path.type: 5, 0..127),
        *(path:      6, text),
        *(query:     7, text),
        ?(fragment:  8, text)]
```

Example:

```
[1, "coap", 3, h'20010DB8000000000000000000000001', 4, 5683, 6,
".well-known", 6, "core"]

[5, 0, 6, ".well-known", 6, "core", 7, "rt=temperature-c"]
```

4. Python

In Python, a sequence of options is encoded as a list of tuples, where each tuple contains one option number and one option value.

The following Python 3.6 code illustrates how to check a sequence of options for being well-formed, absolute, and relative.

<CODE BEGINS>

```
import enum

class Option(enum.IntEnum):
  _BEGIN = 0
  SCHEME = 1
  HOST_NAME = 2
  HOST_IP = 3
  PORT = 4
  PATH_TYPE = 5
  PATH = 6
  QUERY = 7
  FRAGMENT = 8
  _END = 9
```

```
class PathType(enum.IntEnum):
  ABSOLUTE_PATH = 0
  APPEND_RELATION = 1
  APPEND_PATH = 2
  RELATIVE_PATH = 3
  RELATIVE_PATH_1UP = 4
  RELATIVE_PATH_2UP = 5
  RELATIVE_PATH_3UP = 6
  RELATIVE_PATH_4UP = 7

_TRANSITIONS = ([Option.SCHEME, Option.HOST_NAME, Option.HOST_IP,
    Option.PORT, Option.PATH_TYPE, Option.PATH, Option.QUERY,
    Option.FRAGMENT, Option._END],
  [Option.HOST_NAME, Option.HOST_IP],
  [Option.PORT],
  [Option.PORT],
  [Option.PATH, Option.QUERY, Option.FRAGMENT, Option._END],
  [Option.PATH, Option.QUERY, Option.FRAGMENT, Option._END],
  [Option.PATH, Option.QUERY, Option.FRAGMENT, Option._END],
  [Option.QUERY, Option.FRAGMENT, Option._END],
  [Option._END])

def is_well_formed(href):
  previous = Option._BEGIN
  for option, _ in href:
    if option not in _TRANSITIONS[previous]:
      return False
    previous = option
  if Option._END not in _TRANSITIONS[previous]:
    return False
  return True

def is_absolute(href):
  return is_well_formed(href) and \
    (len(href) != 0 and href[0][0] == Option.SCHEME)

def is_relative(href):
  return is_well_formed(href) and \
    (len(href) == 0 or href[0][0] != Option.SCHEME)
```

<CODE ENDS>

Examples:

```
[(Option.SCHEME, "coap"), (Option.HOST_IP, b"\x20\x01\x0D\xB8\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01"), (Option.PORT,
5683), (Option.PATH, ".well-known"), (Option.PATH, "core")]
```

```
[(Option.PATH_TYPE, PathType.ABSOLUTE_PATH), (Option.PATH, ".well-
known"), (Option.PATH, "core"), (Option.QUERY, "rt=temperature-
c")]
```

4.1. Reference Resolution

The following Python 3.6 code defines how to resolve a sequence of options that might be relative to a given base.

<CODE BEGINS>

```
def resolve(base, href, relation=0):
  if not is_absolute(base) or not is_well_formed(href):
    return None
  result = []
  option = Option.FRAGMENT
  if len(href) != 0:
    option = href[0][0]
  if option == Option.HOST_IP:
    option = Option.HOST_NAME
  elif option == Option.PATH_TYPE:
    type = href[0][1]
    href = href[1:]
  elif option == Option.PATH:
    type = PathType.RELATIVE_PATH
    option = Option.PATH_TYPE
  if option != Option.PATH_TYPE or type == PathType.ABSOLUTE_PATH:
    _copy_until(base, result, option)
  else:
    _copy_until(base, result, Option.QUERY)
    if type == PathType.APPEND_RELATION:
      _append_and_normalize(result, Option.PATH, str(relation))
    while type > PathType.APPEND_PATH:
      if len(result) == 0 or result[-1][0] != Option.PATH:
        break
      del result[-1]
      type -= 1
  _copy_until(href, result, Option._END)
  _append_and_normalize(result, Option._END, None)
  return result

def _copy_until(input, output, end):
  for option, value in input:
    if option >= end:
      break
    _append_and_normalize(output, option, value)

def _append_and_normalize(output, option, value):
```

```
        if option > Option.PATH:
          if len(output) >= 2 and \
              output[-1] == (Option.PATH, '') and (
              output[-2][0] < Option.PATH_TYPE or (
              output[-2][0] == Option.PATH_TYPE and
              output[-2][1] == PathType.ABSOLUTE_PATH)):
            del output[-1]
          if option > Option.FRAGMENT:
            return
        output.append((option, value))
```

<CODE ENDS>

4.2. URI Recomposition

The following Python 3.6 code defines how to recompose a URI from an absolute sequence of options.

<CODE BEGINS>

```
      def recompose(href):
        if not is_absolute(href):
          return None
        result = ''
        no_path = True
        first_query = True
        for option, value in href:
          if option == Option.SCHEME:
            result += value + ':'
          elif option == Option.HOST_NAME:
            result += '//' + _encode_reg_name(value)
          elif option == Option.HOST_IP:
            result += '//' + _encode_ip_address(value)
          elif option == Option.PORT:
            result += ':' + _encode_port(value)
          elif option == Option.PATH:
            result += '/' + _encode_path_segment(value)
            no_path = False
          elif option == Option.QUERY:
            if no_path:
              result += '/'
              no_path = False
            result += '?' if first_query else '&'
            result += _encode_query_argument(value)
            first_query = False
          elif option == Option.FRAGMENT:
            if no_path:
              result += '/'
```

```
            no_path = False
          result += '#' + _encode_fragment(value)
      if no_path:
        result += '/'
        no_path = False
      return result

    def _encode_reg_name(s):
      return ''.join(c if _is_reg_name_char(c)
                     else _encode_pct(c) for c in s)

    def _encode_ip_address(b):
      if len(b) == 4:
        return '.'.join(str(c) for c in b)
      elif len(b) == 16:
        return '[' + ... + ']'  # see RFC 5952

    def _encode_port(p):
       return str(p)

    def _encode_path_segment(s):
      return ''.join(c if _is_segment_char(c)
                     else _encode_pct(c) for c in s)

    def _encode_query_argument(s):
      return ''.join(c if _is_query_char(c) and c not in '&'
                     else _encode_pct(c) for c in s)

    def _encode_fragment(s):
      return ''.join(c if _is_fragment_char(c)
                     else _encode_pct(c) for c in s)

    def _encode_pct(s):
      return ''.join('%{0:0>2X}'.format(c) for c in s.encode('utf-8'))

    def _is_reg_name_char(c):
      return _is_unreserved(c) or _is_sub_delim(c)

    def _is_segment_char(c):
      return _is_pchar(c)

    def _is_query_char(c):
      return _is_pchar(c) or c in '/?'

    def _is_fragment_char(c):
      return _is_pchar(c) or c in '/?'

    def _is_pchar(c):
```

```
        return _is_unreserved(c) or _is_sub_delim(c) or c in ':@'

      def _is_unreserved(c):
        return _is_alpha(c) or _is_digit(c) or c in '-._~'

      def _is_alpha(c):
        return c in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' + \
                    'abcdefghijklmnopqrstuvwxyz'

      def _is_digit(c):
        return c in '0123456789'

      def _is_sub_delim(c):
         return c in '!$&\'()*+,;='
```

<CODE ENDS>

4.3. CoAP Encoding

The following Python 3.6 code illustrates how to construct CoAP options from an absolute sequence of options. For simplicity, the code does not omit CoAP options with their default value.

<CODE BEGINS>

```
      def coap(href, to_proxy=False):
        if not is_absolute(href):
          return None
        result = b''
        previous = 0
        for option, value in href:
          if option == Option.SCHEME:
            pass
          elif option == Option.HOST_NAME:
            opt = 3  # Uri-Host
            val = value.encode('utf-8')
            result += _encode_coap_option(opt - previous, val)
            previous = opt
          elif option == Option.HOST_IP:
            opt = 3  # Uri-Host
            if len(value) == 4:
              val = '.'.join(str(c) for c in value).encode('utf-8')
            elif len(value) == 16:
              val = b'[' + ... + b']'  # see RFC 5952
            result += _encode_coap_option(opt - previous, val)
            previous = opt
          elif option == Option.PORT:
            opt = 7  # Uri-Port
```

```
            val = value.to_bytes((value.bit_length() + 7) // 8, 'big')
            result += _encode_coap_option(opt - previous, val)
            previous = opt
          elif option == Option.PATH:
            opt = 11  # Uri-Path
            val = value.encode('utf-8')
            result += _encode_coap_option(opt - previous, val)
            previous = opt
          elif option == Option.QUERY:
            opt = 15  # Uri-Query
            val = value.encode('utf-8')
            result += _encode_coap_option(opt - previous, val)
            previous = opt
          elif option == Option.FRAGMENT:
            pass
        if to_proxy:
          (option, value) = href[0]
          opt = 39  # Proxy-Scheme
          val = value.encode('utf-8')
          result += _encode_coap_option(opt - previous, val)
          previous = opt
        return result

      def _encode_coap_option(delta, value):
        length = len(value)
        delta_nibble = _encode_coap_option_nibble(delta)
        length_nibble = _encode_coap_option_nibble(length)
        result = bytes([delta_nibble << 4 | length_nibble])
        if delta_nibble == 13:
          delta -= 13
          result += bytes([delta])
        elif delta_nibble == 14:
          delta -= 256 + 13
          result += bytes([delta >> 8, delta & 255])
        if length_nibble == 13:
          length -= 13
          result += bytes([length])
        elif length_nibble == 14:
          length -= 256 + 13
          result += bytes([length >> 8, length & 255])
        result += value
        return result

      def _encode_coap_option_nibble(n):
        if n < 13:
          return n
        elif n < 256 + 13:
          return 13
```

```
        elif n < 65536 + 256 + 13:
          return 14
```

<CODE ENDS>

5. Security Considerations

Parsers must operate on input that is assumed to be untrusted. This means that parsers MUST fail gracefully in the face of malicious inputs. Additionally, parsers MUST be prepared to deal with resource exhaustion (e.g., resulting from the allocation of big data items) or exhaustion of the call stack (stack overflow). See Section 8 of RFC 7049 [RFC7049] for security considerations relating to CBOR.

The security considerations discussed in Section 7 of RFC 3986 [RFC3986] also apply to Constrained Resource Identifiers.

6. IANA Considerations

This document has no IANA actions.

7. References

7.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <https://www.rfc-editor.org/info/rfc7049>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/info/rfc8610>.

7.2. Informative References

[RFC5952] Kawamura, S. and M. Kawashima, "A Recommendation for IPv6 Address Text Representation", RFC 5952, DOI 10.17487/RFC5952, August 2010, <https://www.rfc-editor.org/info/rfc5952>.

[RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <https://www.rfc-editor.org/info/rfc7228>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <https://www.rfc-editor.org/info/rfc7252>.

[RFC8288] Nottingham, M., "Web Linking", RFC 8288, DOI 10.17487/RFC8288, October 2017, <https://www.rfc-editor.org/info/rfc8288>.

[W3C.REC-html52-20171214] Faulkner, S., Eicholz, A., Leithead, T., Danilo, A., and S. Moon, "HTML 5.2", World Wide Web Consortium Recommendation REC-html52-20171214, December 2017, <https://www.w3.org/TR/2017/REC-html52-20171214>.

Acknowledgements

Thanks to Christian Amsuess, Ari Keranen, and Dave Thaler for helpful comments and discussions that have shaped the document.

Author's Address

Klaus Hartke
Ericsson
Torshamnsgatan 23
Stockholm SE-16483
Sweden

Email: klaus.hartke@ericsson.com