SCHC Working Group
Internet-Draft
Intended status: Standards Track
Expires: 26 July 2026

L. Corneo
E. Ramos
J. Jimenez
Ericsson

22 January 2026

SCHC Payload compression
draft-corneo-schc-compress-payload-01

Abstract

This document describes several techniques to enable utilization of the same engine to compress and decompress headers from the existing SCHC framework [RFC8724], but used to compress payload of specific protocols. The first approach is to introduce new type of static rules that enable encoding application data. This extensions provides compact and generic variation on how data is organized. The second approach provides dynamic compression and decompression. Here, the system identifies parts of the payload that can be compressed, and enables a SCHC decompressor to restore the original packet.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-corneo-schc-compress-payload/.

Discussion of this document takes place on the schc Working Group mailing list (mailto:schc@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/schc/. Subscribe at https://www.ietf.org/mailman/listinfo/schc/.

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/.

Copyright Notice

Table of Contents

1. Introduction

The main purpose of this document is to describe methods that enable the reduction of the size of the payloads by utilizing the same compression and decompression machinery provided by the SCHC framework [RFC8724]. Utilizing additional steps or introducing new rules, the available SCHC machinery can be reused in IoT devices to also provide compression of the payload.

For the static approach, the introduction of new type of rules that are suitable for the protocols that utilize certain type of time series serialization where certain values can be compressed based on the names, metadata related to each record, value type and base values. By utilizing the compression residue, the variable parts of the fields can be managed.

The dynamic approach is to produce a context that is customized with rules for the type of values that are expected to be delivered by the devices that utilize key-value based data, e.g., JSON [RFC8259], to generate a dictionary that can be later used to encode the data. This is done by inspecting the values that the devices produces by a fix amount of time, and then mapping some of the values to much smaller values that represent the data sent by the device. This reduce the transmission time and hence also the energy required for the transmission.

1.1. Terminology

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 [BCP14] (RFC2119) (RFC8174) when, and only when, they appear in all capitals, as shown here.

This specification requires readers to be familiar with the terms and concepts that are discussed in [RFC7252].

Template-based compression: Payload compresion utilizing a pre-shared model that only requires the values that are changing in the payload.

Common indicator: The common idicators provide a way to compress values that are not necessary to be exact or the value can be dtermined by the decompressor.

2. Template-based payload compression utilizing a static context

This section introduces a templated method for enhancing the compression of time series sensor data, such as SenML [RFC8428], by utilizing a template-based approach within the context rules. By employing a template-based strategy, it becomes possible to reconstruct the data structure (e.g., SenML or JSON payload) without the necessity of transmitting the structure with each payload. The examples use JSON representation, but CBOR [RFC8949] representation can be used in practice to achieve smaller template (and hence rule) size.

2.1. Fixed number of records

When a system is transmitting in the payload the same number of records, the following optimizations can be utilized. As shown on Figure 1 entity with SenML base name (bn) "dev1.example.com/" sends two values periodically one temperature value ("temp" with value 20), and one humidity value ("hum" with value 40). It encodes the values in one SenML Pack consisting of two SenML uncompressed Records:

```
[
  {
    "bn": "dev1.example.com/",
    "n": "temp",
    "v": 20
  },
  {
    "n": "hum",
    "v": 40
  }
]
```

Figure 1: SenML Pack with two SenML Records.

A template with compression/decompression rules can be used to reduce the size of the above payload. For instance, variable values are replaced with $n placeholders, where n represents the position of the field value in the compression residue. Applying the template to the above example will then look as follows:

```
[
  {
    "bn": "dev1.example.com/",
    "n": "temp",
    "v": "$1"
  },
  {
    "n": "hum",
    "v": "$2"
  }
]
```

In this case, upon applying the rule, the compression residue includes exactly two values. The result of the decompression is exactly one SenML Pack with two SenML records.

2.2. Varying number of compression residue values

In some cases, the payload may include values referring to different resource paths, and the number of such measurements may not be known in advance. Furthermore, the measurements may be performed at different times, but sent together in one payload. For instance, the example below shows a payload with multiple temperature and humidity measurements.

```
[
  {
    "n": "dev1.example.com/temp",
    "v": 20,
    "bt": 1696231409
  },
  {
    "n": "dev1.example.com/hum",
    "v": 40
  },
  {
    "n": "dev1.example.com/temp",
    "v": 22,
    "bt": 1696231410
  },
  {
    "n": "dev1.example.com/hum",
    "v": 44
  }
]
```

The template for the compression-decompression rule would look as follows:

```
[
  $repeat(
    {
      "n": "dev1.example.com/temp",
      "v": "$1",
      "bt": "$3"
    },
    {
      "n": "dev1.example.com/hum",
      "v": "$2"
    }
  )
]
```

Figure 2: SenML Pack with varying number of SenML Records that use compression- decompression template.

The $repeat() command means that the two records for "temp" and "hum" readings are repeated and the first record contains a time value that applies to both records. The compression residue would contain a list of triplets with the measurement values for both, followed by the time stamp. The two records are repeated in the decompressed version as many times as there are triplets in the compression residue, i.e., if compression residue contains three values, the decompressed format will include one SenML Pack with two SenML Records. If the compression residue contains six values, the decompressed format will include one SenML Pack with four SenML Records, and so on.

The decompressor is aware of the $repeat() syntax, thus can generate valid JSON by, for example, adding commas between records or valid CBOR/EXI with proper record delimiters, instead of simply repeating the template string given as the parameter. Note that the template mechanism from Section 2.1 can be used in the case of Section 2.2 on Figure 2 as well, but with the $repeat() command, the size of the rule becomes smaller and allows for multiple residue values to be sent, which is especially useful if the number of measurements can vary or is not a priori.

2.3. Additional operations for improved compression

To achieve further compression, the use of absolute timestamps (since epoch) or absolute measurement value may be undesirable and relative timestamps / values might be preferable. To achieve this, the epoch (or base) value needs to be established. The following sections provide different mechanisms through which the epoch / base values can be defined in the template.

2.3.1. Base value defined in the template

In this case, the base value is defined in the template with a "relValue" keyword in the placeholder. Using the example from Figure 2, the template can be updated to the following:

```
[
  $repeat(
    {
      "n": "dev1.example.com/temp",
      "v": $1(relValue:20),
      "bt": $3(relValue:1696231409)
    },
    {
      "n": "dev1.example.com/hum",
      "v": $2(relValue:40)
    }
  )
]
```

Based on the above template, the compression residue values would be relative to the values specified in the template.

2.3.2. Base value based on common indicators

* Timestamps

* Environmental knowledge

In some cases, a shared context can be established without specifying exact relative values. In the case of timestamps, a relTime:x attribute is proposed, where x can be "m" (minute), "h" (hour), "d" (day), "M" (month) and "Y" (year). Hence, without specifying an absolute value, the compressor can send the time relative to the "top of" the minute, hour, day, month, or year. In this way, the neither the rules need to be updated nor a lookup needs to be performed.

In other cases, the shared context can be established through environment conditions. For example, in the case of temperature and humidity, it can be defined that the base value in summer for temperature is 20 and humidity is 60 and in winter the base value is 0 and humidity 10. Such context is established outside the SCHC template but can be expressed in the template using the notation relContext:x where x can mean anything. For instance, x could be "season" using the example above and, when the switch between summer and winter occurs, it is treated as context specific and not expressed in the template.

2.3.3. Variable-length and fixed-length residue values.

SCHC [RFC8724] describes mechanisms for handling both fixed and variable length residue fields. Fixed length fields are specified in the FL field of the rule, whereas the varying length-fields are specified such that the compression residue includes the field length and the value.

The use of varying length fields for the values and the field length needs to be usually included within the residue. However, it may be undesirable to include the field length for static cases. Hence, a fl:x template convention is proposed, e.g., $1(fl:16) which indicates that the length of the $1 is always 16 bits. Hence the draft supports compression even in the case of fixed length field. The special case of all fields using the same length can be handled by indicating that length in the FL field.

2.3.4. Mapping the template-approach to the SCHC C/D Context

Rules in SCHC are described using the Compression/Decompression Context using the format (and terminology) based on the diagram in Figure 6 of Section 7.1 of [RFC8724]).

This section describes the values for the C/D context when using the template-based approach.

FID:
: A new Field ID called "StructuredAppData" is introduced to indicate that the compression/decompression is being performed on the data / payload.

FL:
: If all compression residue values are of same length (e.g., 4 bits), this field contains the bit length. Otherwise, this field contains value 0 indicating that variable-length values are used in the compression residue.

FP:
: set to 1 (default)

DI:
: As per the current spec (i.e., it can be Up, Dw, or Bi).

Target Value (TV):
: The Target Value is where the template itself is defined.

Matching Operator (MO):

A new MO called "template" is introduced to indicate that the compressor/decompressor should follow template based matching operations.

Compression/Decompression Action (CDA):
A new action called "template" is introduced. The compressor sends all the values (as described in examples above) and the decompressor reconstructs the original payload but replacing the $x placeholders (and expanding the other operations such as $repeat, etc.).

3. Dynamic compression of Payload

The dynamic payload compression requires a new SCHC context that is updated to the compressor after inspecting packets' payloads. This can be done in three phases where the first phase collects statistics about the payload, the second phase identifies the payload fields that may be compressed, and the third phase generates a new SCHC context with rules targeting such fields. This dynamic compression is intended to work with key-value based payload (e.g., JSON).

The SenML Record below is the reference payload for the whole section.

```
[
  {
    "bn": "2001:db8:1234:5678::1/",
    "n": "temperature",
    "u": "Cel",
    "v": 25.2
  },
  {
    "n": "humidity",
    "u": "%RH",
    "v": 30
  }
]
```

Figure 3: Reference example of IoT payload in SenML format.

3.1. Payload analysis

In the first phase, the decompressor inspects the payload carried through the SCHC packet. All the keys and values of the key-value pairs are stored in a map-like, or list-like, data structure. For example, if the payload contains the key-value pair {"n": "temperature"} the associated data structure may be a 'map' or a 'list'.

map:
Below is an example of a map-like data structure that records each key-value pair from the payload along with the count of how many times each key or value appears, e.g., {"n": 1, "temperature": 1}.

list:
Alternatively, a list-like data structure can be used to store only the keys and values from the payload's key-value pairs, such as ["n", "temperature"].

Additional methods for storing payload information may be used, e.g., a timestamp of the received payload. The information recorder in this phase mainly depends on the selection of the fields that are going to be compressed.

3.2. Fields selection

After payload analysis on Section 3.1, the decompressor entity manipulates the data structure to create an unique key for each value.

The key creation mechanism involves encoding different pieces of information in a structured format. This is achieved by separating components with a special character, such as a dot. For instance, consider the format info1.info2.

The segment before the dot represents a standard content type, such as application/json+senml. The segment after the dot includes the name of the payload field, which is the key in a key-value pair, followed by a unique identifier number, like n1, to ensure uniqueness. Consequently, a key might be formatted as application/senml+json.n1.

This structured key informs the decompressor entity that: (i) the payload is compressed using JSON-encoded SenML, (ii) each field should be decompressed into a "key": "value" format, (iii) fields sharing the same numeric identifier, such as n1, should be enclosed within curly brackets, and (iv) the entire structure should be wrapped in square brackets (as per [RFC8428]).

3.2.1. Compression triggers

The criteria for compressing key-value pairs can differ based on specific use cases. This section outlines some triggers that determine whether a key-value pair should be compressed, such as:

Number of occurrences:

A counter keeps track of how many instances of a specific key-value pair have been received in adjacent messages. The payload compression may be configured to trigger compression for all the key-value pairs that have been sent at least 5 times in adjacent messages.

Key length:
The internal state of the decompressor entity keeps track of the length of the key-value pairs and, based on a predefined threshold, e.g., 100 bytes, decides on whether the pair is to be compressed or not.

Complex triggers:
A condition accounting for multiple variables can be defined; for example, a timer after which all the key-value pairs have appeared in the payload more than 5 times are to be selected for compression.

3.3. SCHC context generation

Following the field selection process in Section 3.2, the decompressor SHALL generate a SCHC context with rules targeting the selected key-value pairs. For each pair, the decompressor SHALL select SCHC rule fields that align with its policies, such as minimizing payload length and using the CDA "elide" where applicable.

While payload size reduction is a common goal, other objectives, such as incorporating redundancy to reduce transmission errors, MAY also be considered.

3.4. Generate SCHC payload compression rules

The server produces SCHC payload compression rules like the ones showed in Figure 4. In SenML, JSON, and key-value data structures at large, only two fields can be directly mapped to a SCHC rule, namely, FID (the key) and TV (the target value). However, a SCHC rule is composed by 7 to 9 fields. Such fields are populated according to specific needs of the client. As an example, the configuration may be set to operate in a way to minimize the transmitted payload, rather than minimize the overhead of storing the SCHC context for each client. Next, it is explained an example of how the SCHC payload rules generation could be implemented. Rules can be mapped as following:

FID (Field Identifier):
The Field Identifier (FID) can be directly derived from the keys shown in Figure 4, such as application/json+senml.v.1. In this example, the key’s structure carries semantics. The segment

before the delimiter character specifies the payload's content type, guiding the decompressor on how to handle the compressed key-value pair. The segment after the delimiter includes the key's name in the key-value pair, followed by a number that groups related fields. For instance, FIDs with the number 1, like n.1, u.1, and v.1, are part of the same SenML record and will be enclosed in curly brackets.

FL (Field Length):
The FL is computed by the system evaluating the size of the value to compress.

FP (Field Position):
In the case of SenML, the FP is not useful, meaning that the placement of the field would mostly follow the same pattern. However, if for any specific reason the position of the field is of relevance, this field can be used.

DI (Direction):
The DI expresses whether the field for compression/decompression appears at upstream or downstream. In the case of IoT scenarios, the direction is almost always upstream since it is critical to reduce transmission time to extend as much as possible devices' battery life.

TV (Target Value):
The TV is the value on which to perform the Matching Operation (MO). Essentially, it is the value, or part of the value, to be compressed.

MO (Matching Operator):
The MO is the operation that is applied to every payload value and, when there is a match between such value and the TV, a de/compression action (CDA) will be applied. In a nutshell, if there is a match, the payload value will be de/compressed.

CDA (Compression/Decompression Action):
This is the action to be taken to the payload value in case of matching with the TV. Common actions are not-sent (the value will be omitted in the sent payload), value-sent (the value will be sent), L(ess)S(significant)B(it) (only the least significant part of the payload value will be sent).

Example:

```
   {
     "ruleID": 12,
     "ruleLength": 4,
     "compression": [
       {
         "FID": "application/senml+json.bn.1",
         "FL": 22,
         "FP": 1,
         "DI": "Up",
         "TV": "2001:db8:1234:5678::1/",
         "MO": "equal",
         "CDA": "not-sent"
       },
       {
         "FID": "application/senml+json.n.1",
         "FL": 11,
         "FP": 2,
         "DI": "Up",
         "TV": "temperature",
         "MO": "MSB",
         "MOa": "10",
         "CDA": "LSB"
       },
       {
         "FID": "application/senml+json.n.2",
         "FL": 8,
         "FP": 3,
         "DI": "Up",
         "TV": "humidity",
         "MO": "MSB",
         "MOa": "7",
         "CDA": "LSB"
       },
       {
         "FID": "application/senml+json.u.1",
         "FL": 3,
         "FP": 4,
         "DI": "Up",
         "TV": "Cel",
         "MO": "equal",
         "CDA": "not-sent"
       },
       {
         "FID": "application/senml+json.u.2",
         "FL": 33,
         "FP": 5,
         "DI": "Up",
         "TV": "%RA",
```

```
        "MO": "equal",
        "CDA": "not-sent"
      },
      {
        "FID": "application/senml+json.v.1",
        "FL": 4,
        "FP": 6,
        "DI": "Up",
        "TV": "",
        "MO": "ignore",
        "CDA": "value-sent"
      },
      {
        "FID": "application/senml+json.v.2",
        "FL": 4,
        "FP": 7,
        "DI": "Up",
        "TV": "",
        "MO": "ignore",
        "CDA": "value-sent"
      }
    ]
  }
```

Figure 4: Example of SCHC rule generated in the dynamic payload compression approach.

In the above rule, the fields application/senml+json.u.1/2 are never sent because are known a-priori and will probably never change (a temperature sensor will always return Celsius or Fahrenheit). Fields application/senml+json.n.1/2 are shortened only to their last character due to MSB and LSB, e.g., e and y. Finally, application/senml+json.v.1/2, are ignored by the compressor and will be sent without compression, given that they may change. After the dynamic payload compression, the transmitted payload may look as follows:

0x0C657941C9999A0000001E

In the above example, 0C is the hexadecimal representation of 12, which indicates that whatever follows is compressed according to rule 12. That is, the SenML base name, 2001:db8:1234:5678::1/, is omitted. Then, the last letter of the names is compressed, namely, 65 and 79, representing ASCII for e (from temperature) and y (from humidity). Moreover, the measurement units Cel and %RH are omitted (not-sent CDA applies). Finally, the remaining values are encoded as 4 bytes.

3.4.1. Encoding decompression hints in FID

This section describes how to manipulate the FID field so to provide hints to the decompressor on how to reconstruct the original payload.

As an example, the content-type of the payload, the name of the field associated with the value, and a group identifier may be encoded in the FID. It is possible to separate these pieces of information using a delimiter, e.g., a single character such as a dot. For example, the left part of the first dot is a standard content type, e.g., application/json+senml, while the right part of the first dot, the name of the payload field followed by a number used to group fields together, e.g., n.1. As a result, the name of such a key would be application/senml+json.n.1.

This way, the decompressor knows that the compressed payload is of SenML type encoded in JSON, thus knows that every field must be decompressed as "key": "value", every field with the same numeric identifier, e.g., n.1, u.1 and v.1, enclosed by curly brackets, and finally everything enclosed into square brackets, as per SenML encoding. More generally, the hints for the decompressor are encoded as: <content-type>.<key>.<group-number>

3.4.2. Decompression of SCHC compressed payload

The SCHC decompressor starts by reading the first byte, 0C, which indicates the SCHC RuleID, 12. It retrieves and applies the rules sequentially. The first rule, application/senml+json.bn.1, has a CDA of not-sent, meaning the bn value was omitted during compression and must be added back. The decompressor reconstructs this as "bn": "2001:db8:1234:5678::1/".

Next, for the rule application/senml+json.n.1, the decompressor identifies an MSB CDA, indicating a 1-byte value was added during compression. It retrieves 0x65 (representing "E") and appends it to the target value, forming the JSON pair: "n": "temperature". This process is repeated for the remaining existing rules.

The decompressor must know the payload’s content-type to accurately reconstruct the packet. While this example uses SenML JSON, other formats like YAML or XML can also be supported for greater flexibility.

3.5. Analysis of payload data and selection of values to compress

There may be several ways to build the compression map exchanged between the server and the client. This section provides one such instance based on a threshold. That is, when a value is repeated more than a predefined number of times (3 in this particular example), it is considered redundant and will be compressed. More elaborated techniques may be used, e.g., size of a value.

The server inspects the payload sent from the devices. For every message, in key-value formats, e.g., JSON, the server counts the occurrences of each value. Considering the payload from the example above, after receiving the payload for the first time, the server produces such map:

```
{
  "2001:db8:1234:5678::1/temperature": 1,
  "2001:db8:1234:5678::1/humidity": 1,
  "Cel": 1,
  25.2: 1,
  "%RH": 1,
  30: 1
}
```

The above map counts how many times a particular value has been sent as payload. For example, after 3 iterations the map may look as follows:

```
{
  "2001:db8:1234:5678::1/temperature": 3,
  "2001:db8:1234:5678::1/humidity": 3,
  "Cel": 3,
  25.2: 1,
  25.1: 1,
  24.8: 1,
  "%RH": 3,
  30: 1,
  31: 1,
  35: 1
}
```

In the above map, the values of the temperature (25.2, 25.1, 24.8) and humidity (30, 31, 35) have changed over time, and their counter is set to 1. However, the sensor names and measurement units have not changed and their counter value increased to 3. At this point, the server detects that some values in the payload are redundant (sent more than 3 times) and could be compressed by assigning to them a single-digit character. As a result, the server builds and share with the client the following compression map:

```
{
  0: "2001:db8:1234:5678::1/temperature",
  1: "2001:db8:1234:5678::1/humidity",
  2: "Cel",
  3: "%RH",
}
```

3.6. Updating the compression map

When the server identifies another value to be redundant, the sender will send another compression map including only the new element (associated to a new unique number). Assuming that the temperature has been constant (25 Celsius) for the last three updates, the new map will look as follows:

```
{
  4: 25
}
```

If an element is no longer used, it can be removed from the compression map. For instance, if a sensor’s name changes, the old name can be invalidated, allowing the client to free memory by deleting it. To remove the value 2001:db8:1234:5678::1/temperature with ID 0, assign a negative value to the ID:

```
{
    0: -1
}
```

4. IANA Considerations

This document has the following actions for IANA.

Note to RFC Editor: Please replace all occurrences of "[RFC-XXXX]" with the RFC number of this specification and delete this paragraph.

5. Normative References

[BCP14] Best Current Practice 14,
<https://www.rfc-editor.org/info/bcp14>.
At the time of writing, this BCP comprises the following:

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>.

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>.

[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/rfc/rfc7252>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/rfc/rfc8259>.

[RFC8428] Jennings, C., Shelby, Z., Arkko, J., Keranen, A., and C. Bormann, "Sensor Measurement Lists (SenML)", RFC 8428, DOI 10.17487/RFC8428, August 2018, <https://www.rfc-editor.org/rfc/rfc8428>.

[RFC8724] Minaburo, A., Toutain, L., Gomez, C., Barthel, D., and JC. Zuniga, "SCHC: Generic Framework for Static Context Header Compression and Fragmentation", RFC 8724, DOI 10.17487/RFC8724, April 2020, <https://www.rfc-editor.org/rfc/rfc8724>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/rfc/rfc8949>.

Appendix A. Acknowledgements

A great token of appreciation to Rajat Kandoi and Ari Kernen for their support in the inception and review of this draft. Their ideas and thinking are also reflected in the text of the draft.

Authors' Addresses

Lorenzo Corneo
Ericsson
Email: lorenzo.corneo@ericsson.com

Edgar Ramos
Ericsson
Email: edgar.ramos@ericsson.com

Jaime Jimenez
Ericsson
Email: jaime.jimenez@ericsson.com