Workgroup: Network Working Group
Internet-Draft:
draft-mcquistin-augmented-ascii-diagrams-13
Published: 22 October 2023
Intended Status: Experimental
Expires: 24 April 2024
Authors: S. McQuistin, University of St Andrews
V. Band, University of Glasgow
D. Jacob, University of Glasgow
C. S. Perkins, University of Glasgow

Describing Protocol Data Units with Augmented Packet Header Diagrams

Abstract

This document describes a machine-readable format for specifying the syntax of protocol data units within a protocol specification, known as an Augmented Packet Header Diagram. This comprises a consistently formatted packet header diagram, using a well-defined syntax, that is followed by structured explanatory text. The Augmented Packet Header Diagram format is designed to maintain human readability, while also supporting automated parser generation from protocol specification documents. This document is itself an example of how the format can be used.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Packet header diagrams are widely used for describing the syntax of binary protocols. In otherwise largely textual documents, they allow for the visualisation of packet formats in a readily understandable manner, aiding the implementation of parsers for the protocols that they describe.

Figure 1 gives an example of how packet header diagrams are used to define binary protocol formats. The format has an obvious structure: the diagram clearly delineates each field, showing its width and its position within the header. This type of diagram is designed for human readers, but is consistent enough that it should be possible to develop a tool that generates a parser for the packet format from the diagram.

```
:    0                   1                   2                   3
:    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
:   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:   |          Source Port          |       Destination Port        |
:   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:   |                        Sequence Number                        |
:   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:   |                    Acknowledgment Number                      |
:   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:   |  Data |           |U|A|P|R|S|F|                               |
:   | Offset| Reserved  |R|C|S|S|Y|I|            Window             |
:   |       |           |G|K|H|T|N|N|                               |
:   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:   |           Checksum            |         Urgent Pointer        |
:   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:   |                    Options                    |    Padding    |
:   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:   |                             data                              |
:   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1: The TCP packet header format (from *[RFC793]*)

Unfortunately, the format of such packet diagrams varies both within and between documents in the RFC series. This variation makes it difficult to build tools to generate parsers from the specifications. Better tooling could be developed if protocol specifications adopted a consistent format for their packet descriptions. Indeed, this underpins the format described by this draft: we want to retain the benefits that packet header diagrams provide, while identifying the benefits of adopting a consistent format.

This document describes a consistent packet header diagram format and accompanying structured text phrases, collectively known as Augmented Packet Header Diagrams, that fully specify the syntax of protocol data units allowing for the automated generation of packet parsing code. Broad design principles, that maintain the primacy of human readability and flexibility in writing, are described, before the format itself is given.

This document is itself an example of the approach that it describes, with the packet header diagrams and structured text format described by example. Packet header diagrams that do not use the Augmented Packet Header Diagram format are marked by a colon at the beginning of each line, as in Figure 1; this prevents them from being parsed by the accompanying tooling.

The goal of this document is to define a format that, with minimal effort on the part of authors, provides benefits to the users of those documents that adopt it. An explicit non-goal of this work, as will be further discussed in Section 4, is to be prescriptive about how the format should be adopted. For example, alterations to the underlying XML format of RFCs are out-of-scope.

This draft describes early work. As consensus builds around the particular syntax of the format described, a formal ABNF specification (Appendix A) will be provided. Code that parses documents written using this format, and that automatically generates parser code for the described protocols, is described in Appendix B.

2. Background

We begin by considering how packet header diagrams are currently used within the RFC series. This exposes the limitations that the current usage has in terms of machine-readability, guiding the design of the augmented packet header format that we propose.

While this document focuses on machine-readability of packet header diagrams, this section also discusses the use of other structured or formal languages within IETF documents. Considering how and why these languages are used provides an instructive contrast to the relatively incremental approach proposed here.

2.1. Limitations of Current Packet Header Diagrams

Packet header diagrams are frequently used in IETF standards to describe the format of binary protocols. While there is no standard for how these diagrams should be formatted, they have a broadly similar structure, where the layout of a protocol data unit (PDU) or structure is shown in diagrammatic form, followed by a description of the fields that it contains. An example of this format, taken from a draft of the QUIC specification, is given in Figure 2.

```
:   The RESET_STREAM frame is as follows:
:
:    0                   1                   2                   3
:    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
:   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:   |                        Stream ID (i)                        ...
:   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:   |  Application Error Code (16)  |
:   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:   |                        Final Size (i)                       ...
:   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:
:   RESET_STREAM frames contain the following fields:
:
:   Stream ID:  A variable-length integer encoding of the Stream ID
:      of the stream being terminated.
:
:   Application Protocol Error Code:  A 16-bit application protocol
:      error code (see Section 20.1) which indicates why the stream
:      is being closed.
:
:   Final Size: A variable-length integer indicating the final size
:      of the stream by the RESET_STREAM sender, in unit of bytes.
```

Figure 2: QUIC's RESET_STREAM frame format (from *[QUIC-TRANSPORT]*)

These packet header diagrams, and the accompanying descriptions, are formatted for human readers rather than for automated processing. As a result, while there is rough consistency in how packet header diagrams are formatted, there are a number of limitations that make them difficult to work with programmatically:

Inconsistent syntax: There are two classes of consistency that are needed to support automated processing of specifications: internal consistency within a diagram or document, and external consistency across all documents.

Figure 2 gives an example of internal inconsistency. Here, the packet diagram shows a field labelled "Application Error Code", while the accompanying description lists the field as "Application Protocol Error Code". The use of an abbreviated name is suitable for human readers, but makes parsing the structure difficult for machines. Figure 3 gives a further example, where the description includes an "Option-Code" field that does not appear in the packet diagram; and where the description states that each field is 16 bits in length, but the diagram shows the OPTION_RELAY_PORT as 13 bits, and Option-Len as 19 bits. Another example is [RFC6958], where the packet format diagram showing the structure of the Burst/Gap Loss Metrics Report Block shows the

Number of Bursts field as being 12 bits wide but the corresponding text describes it as 16 bits.

Comparing Figure 2 with Figure 3 exposes external inconsistency across documents. While the packet format diagrams are broadly similar, the surrounding text is formatted differently. If machine parsing is to be made possible, then this text must be structured consistently.

Ambiguous constraints: The constraints that are enforced on a particular field are often described ambiguously, or in a way that cannot be parsed easily. In Figure 3, each of the three fields in the structure is constrained. The first two fields ("Option-Code" and "Option-Len") are to be set to constant values (note the inconsistency in how these constraints are expressed in the description). However, the third field ("Downstream Source Port") can take a value from a constrained set. This constraint is expressed in prose that cannot readily by understood by machine.

Poor linking between sub-structures: Protocol data units and other structures are often comprised of sub-structures that are defined elsewhere, either in the same document or within a related document (e.g., a base protocol specification document and a set of extensions). Chaining these structures together is essential for machine parsing: the parsing process for a protocol data unit is only fully expressed if all elements can be parsed.

Figure 2 highlights the difficulty that machine parsers have in chaining structures together. Two fields ("Stream ID" and "Final Size") are described as being encoded as variable-length integers; this is a structure described elsewhere in the same document. Structured text is required both alongside the definition of the containing structure and with the definition of the sub-structure to allow a parser to link the two together.

Lack of extension and evolution syntax: Protocols are often specified across multiple documents, either because the protocol explicitly includes extension points (e.g., profiles and payload format specifications in RTP [RFC3550]) or because definition of a protocol data unit has changed and evolved over time. As a result, it is essential that syntax be provided to allow for a complete definition of a protocol's parsing process to be constructed across multiple documents.

```
:   The format of the "Relay Source Port Option" is shown below:
:
:    0                   1                   2                   3
:    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
:   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:   |    OPTION_RELAY_PORT    |         Option-Len                   |
:   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:   |    Downstream Source Port     |
:   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:
:   Where:
:
:   Option-Code:  OPTION_RELAY_PORT. 16-bit value, 135.
:
:   Option-Len:  16-bit value to be set to 2.
:
:   Downstream Source Port:  16-bit value.  To be set by the IPv6
:      relay either to the downstream relay agent's UDP source port
:      used for the UDP packet, or to zero if only the local relay
:      agent uses the non-DHCP UDP port (not 547).
```

Figure 3: The DHCPv6 Relay Source Port Option (from *[RFC8357]*)

2.2. Formal languages in standards documents

A small proportion of IETF standards documents contain structured and formal languages, including ABNF [RFC5234], ASN.1 [ASN1], C, CBOR [RFC7049], JSON, the TLS presentation language [RFC8446], YANG models [RFC7950], and XML. While this broad range of languages may be problematic for the development of tooling to parse specifications, these, and other, languages serve a range of different use cases. ABNF, for example, is typically used to specify text protocols, while ASN.1 is used to specify data structure serialisation. This document specifies a structured language for specifying the parsing of binary protocol data units.

3. Augmented Packet Header Diagrams

As discussed in Section 2.1 there are limitations to how packet header diagrams are used that must be addressed if they are to be parsed by machine. In this section, an augmented packet header diagram format is described. The principles that underpin the design of this format are discussed in Section 4.

The concept is illustrated by example, with accompanying explanatory descriptions. This is appropriate, given the visual nature of the language. A formal specification of the grammar for the augmented packet header diagrams is given in Appendix A.

The augmented packet header diagram format described in this document aligns with an underlying protocol data model, the Network Packet Representation, outlined in [NETWORKING2021]. Using a typed data model to express the syntax of protocols, and the way that they are parsed, allows for safety and complexity properties to be defined and enforced. This underlying representation is independent of the description language used: the Augmented Packet Header Diagram format, defined in the sections that follow, is only one such approach.

Our examples are drawn from the specifications of TCP [RFC9293], STUN [RFC8489], and QUIC [QUIC-TRANSPORT]. These examples are illustrative only of the Augmented Packet Header Diagram format that we define in this document, and they do not necessarily reflect the current state of the specifications they are taken from. For example, the published QUIC specification [RFC9000] does not use packet header diagrams to describe the syntax of the protocol.

3.1. Defining Protocol Data Units

The Augmented Packet Header Diagram format introduces a new protocol data unit description using the exact phrase "A _______ is formatted as follows" within a paragraph (the phrasing "An ______ is formatted as follows" can also be used). Optionally, this phrase can include a note or comment, delimited by commas, immediately following the PDU name. That is, "A/An _______, with an optional comment, is formatted as follows" can be used to introduce a PDU description. This introductory phrase is followed by the PDU description itself, as a packet diagram, starting with a header line to show the bit width of the diagram. The description of the fields follows the diagram, after a paragraph that begins with the text "where:".

PDU names must be unique, both within a document, and across all documents that are linked together using the structured language defined in Section 3.9.

Each field in the PDU is defined by a field in the packet header diagram, a structured text definition, and a prose description. The structured text definition comprises the field name and an optional short name in parenthesis. These are followed by a colon, the field length, an optional expression constraining the value of the field, an optional presence constraint, and, optionally, a terminating period. The prose following the terminating period provides space for optional notes or comments. Field names cannot be the same as a previously defined PDU name, and must be unique within a given structure definition. Prose descriptions may include structured text (e.g., as defined in Section 3.6).

A PDU may be defined not only by the layout and type of its fields, but also by the values of those fields. For example, field values may be constrained to some known value or to be within a range. The "Data Offset" and "Reserved" fields in the example below make use of value constraints. More generally, the augmented packet header diagram format enables a boolean expression to be attached to a field, which must be true for the PDU to be parsed successfully.

In addition, a PDU may contain fields that have a size that is specified in terms of the value of another field. The constraint syntax can be used to specify the length of fields in known units (of bits, bytes, or other structures). In the example below, the "Options" field is defined in units of "TCP Option" structures, and this is indicated by square brackets in the diagram and description list.

If the units are of variable-width, then it may not be possible to specify the length of the sequence. However, it is still necessary to be able to constrain the overall width of the field. To support this, the constraint syntax includes a "size" function that evaluates to the width, in bits, of the given named field. The "Options" field in the example below makes use of this syntax to constrain the size of the field.

Finally, the presence of a field in a PDU may depend on the value of other fields in that PDU. As shown by the "Options" field in the example below, a constraint expression can be attached to each field, where that field is only present in the PDU when the expression is true.

We define an ABNF grammar for constraint expressions in Appendix A.1. This grammar is used across value, size, and presence constraint expressions.

The elements of an augmented packet header diagram can be illustrated using the TCP Header format [RFC9293]. A TCP Header is formatted as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |       |C|E|U|A|P|R|S|F|                               |
| Offset| Rsrvd |W|C|R|C|S|S|Y|I|          Window Size          |
|       |       |R|E|G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           [Options]                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               :
:                            Payload                            :
:                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

where:

Source Port: 16 bits. This is a fixed-width field, whose full label is shown in the diagram. The field's width, 16 bits, is given in this description, separated from the field's label by a colon.

Destination Port: 2 bytes. This is a fixed-width field as previously described. Where fields are an integral number of bytes in size, the field length can be given in bytes rather than in bits.

Sequence Number: 32 bits. This is a fixed-width field as previously described.

Acknowledgment Number: 32 bits. This is a fixed-width field as previously described.

Data Offset (DOffset): 4 bits; DOffset >= 5. This is a fixed-width field, with a constraint on its value. At most one field value constraint may be given per field, and if provided, it must be given as a boolean expression, separated by a semi-colon in the field definition name. If present, a value constraint must follow the name, short name, and length of the field, but appear before any presence constraint, if applicable. The order of the field must be the same in both the diagram and description list.

Reserved (Rsrvd): 4 bits; Rsrvd == 0. This is a fixed-width field, with a value constraint, as previously described. This is a shorter field, whose full label is too large to be shown in the diagram. A short label (Rsrvd) is used in the diagram, and this short label is provided, in brackets, after the full label in the description list.

Control bits: Optionally, field description lists can be nested. In this example, "Control bits" describes the group of 8 single bit fields that are described in the list that follows; it is these single bit fields that will form part of the structure.

CWR: 1 bit. This is a fixed-width field as previously described.

ECE: 1 bit. This is a fixed-width field as previously described.

URG: 1 bit. This is a fixed-width field as previously described.

ACK: 1 bit. This is a fixed-width field as previously described.

PSH: 1 bit. This is a fixed-width field as previously described.

RST: 1 bit. This is a fixed-width field as previously described.

SYN: 1 bit. This is a fixed-width field, with a value constraint, as previously described.

FIN: 1 bit; (FIN == 0) || (SYN == 0). This is a fixed-width field, with a value constraint, as previously described.

Window Size: 16 bits. This is a fixed-width field as previously described.

Checksum: 16 bits. This is a fixed-width field as previously described.

Urgent Pointer: 16 bits. This is a fixed-width field as previously described.

Options: [TCP Option]; size(Options) == (DOffset-5)*32; present only when DOffset > 5.
This is a variable-width field that is comprised of a sequence of TCP Option sub-structures. TCP Option is an enumerated type, to be defined in Section 3.3. As defined, the TCP Option type can be either 2 or 3 bytes, depending on the option type. As a result, it is not possible to specify the number of TCP Option structures that the Option field will contain. However, the overall size of

the field can be constrained. The "size(Options) == (DOffset-5)*32" makes use of the "size" function. This evaluates to the size, in bits, of the named field. The argument passed to the "size" field must be the name of the field being defined, or of a previously defined field.

The "DOffset" field contains the number of 32-bit words that are present in the TCP Header. By default, with no TCP options, this is 5. As a result, the size of the Options field is constrained to the value of DOffset, less 5, and multiplied to get the value in bits.

The presence of the "Options" field is predicated on an expression. Optional fields are indicated by the presence of "; present only when [expr]." at the end of the definition term.

Payload. This is a multi-row variable-length field, denoted in the packet header diagram by the ":" notation in the field's border. The length of the Payload is not specified, and hence needs to be inferred from the total length of the packet and the lengths of the known fields. There can only be one field of unspecified size in a PDU Fields where the length is not specified may also denote this with the phrase "variable length" in place of the length definition.

The simplest PDU is one that contains only a set of fixed-width fields in a known order, with no optional fields or variation in the packet format.

Some packet formats include variable-width fields (e.g., the "Options" field in the example above), where the size of a field is either derived from the value of some previous field, or is unspecified and inferred from the total size of the packet and the size of the other fields.

To ensure that there is no ambiguity, a PDU description can contain only one field whose length is unspecified. The length of a single field, where all other fields are of known (but perhaps variable) length, can be inferred from the total size of the containing PDU. For example, the "Payload" field in the example above is unspecified; its length can be determined by subtracting the length of the other fields from the total size of the PDU.

3.2. Cross-Referencing Previously Defined Fields

Binary formats often reference sub-structures that have been defined earlier in the specification. For example, in TCP [RFC9293], the SACK Range Option (a TCP option type, as will be discussed in Section 3.3) is defined in terms of of SACK blocks. A SACK Block is formatted as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Left Edge                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Right Edge                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

where:

Left Edge: 4 bytes. This is a fixed-width field, as described previously.

Right Edge: 4 bytes. This is a fixed-width field, as described previously.

The SACK Block sub-structure is then used in the definition of the SACK Range Option.

A SACK Range Option is formatted as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       5       |     Length    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            [Blocks]                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

where:

Option Kind (Kind): 1 byte; Kind == 5. This is a fixed-width field, as described previously.

Option Length (Length): 1 byte. This is a fixed-width field, as described previously.

Blocks: (Length-2)/8 SACK Blocks. Where a field is comprised of a sequence of previously defined structures, square brackets can be used to indicate this in the diagram. The length of the sequence can be defined using the constraint expression grammar as described earlier. Where the length is unknown, the type of each element of the sequence must be given in square brackets.

In this example, both a PDU name (SACK Block) and a field name (Length) are used in the constraint expression. This is possible because field names cannot be the same as previously defined PDU names.

3.3. Specifying Enumerated Types

In addition to the use of the sub-structures, it is desirable to be able to define a type that may take the value of one of a set of alternative structures.

The alternative structures that comprise an enumerated type are identified using the exact phrase "The <enumerated type name> is one of: <list of structure names>" where the list of structure names is a comma separated list (with the last element, if there is more than one element, preceded by 'or'), each optionally preceded by "a" or "an". The structure names must be defined within the document or a linked document. Optionally, this phrase can include a note or comment, delimited by commas, immediately following the enumerated type name. That is, "The <enumerated type name>, with an optional comment, is one of: <list of structure names>" can be used to define an enumerated type. In both cases, the colon is optional; for example, "The <enumerated type name> is one of <list of structure names>" is valid.

Where an enumerated type has only two variants, an alternative phrase can be used: "The <enumerated type name> is either a <variant 1 name> or <variant 2 name>". The names of the variants must be defined within the document or a linked document. An optional note or comment can be included with this alternative phrasing: "The <enumerated type name>, with an optional comment, is either a <variant 1 name> or <variant 2 name>" can be used.

An EOL Option is formatted as follows:

```
0
0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|       0       |
+-+-+-+-+-+-+-+-+
```

where:

Option Kind (Kind): 1 byte; Kind == 0. This is a fixed-width field, with a value constraint, as previously described.

A TCP Option is either an EOL Option or a SACK Range Option.

3.4. Splitting Fields

In some binary formats, fields are striped across multiple non-contiguous bits. This is often to allow for backwards compatibility with previous definitions of the same fields in earlier documents: striping in this way allows for careful use of the possible range of values.

This format is illustrated using the STUN Message Type [RFC8489]. A STUN Message Type is formatted as follows:

```
 0                   1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|M|M|M|M|M|C|M|M|M|C|M|M|M|M|
|B|A|9|8|7|1|6|5|4|0|3|2|1|0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

where:

Method (M): 12 bits (split field). This field is comprised of multiple sub-fields (M0 through MB) as shown in the diagram. That these sub-fields should be concatenated, after parsing, into a single field is indicated by their being labelled using the 'M' short field name followed by a single hexadecimal digit, with the least significant bit labelled with 0, and subsequent bits labelled in sequence.

Class (C): 2 bits (split field). This field follows the same format as M described above.

3.5. Extending Sub-Structures

A PDU may not only use or reference existing sub-structures, but they may extend them, adding new fields, or enforcing different or additional constraints.

Where a sub-structure is extended, the diagram may show the sub-structure as a block, labelled with the sub-structure name. It may also be desirable to show the sub-structure diagram in full; in this case, the fields must be given in the same order and be of the same length. New field constraints can be shown. Similarly, in the description list, those fields inherited without change (i.e., with no change to their constraints) do not need to be repeated. Those with different or additional constraints must be described, and the order of the fields in the description list must match that of the sub-structure and the containing structure.

This can be illustrated using QUIC [QUIC-TRANSPORT]. A Long Header is formatted as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+
|1|1| T | R | P |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             Version                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    DCID Len   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 Destination Connection ID (DCID)            ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    SCID Len   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Source Connection ID (SCID)                ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

where:

Header Form (HF): 1 bit; HF == 1. This is a fixed-width field, with a value constraint, as previously described.

Fixed Bit (FB): 1 bit; FB == 1. This is a fixed-width field, with a value constraint, as previously described.

Long Packet Type (T): 2 bits. This is a fixed-width field as previously described.

Reserved Bits (R): 2 bits. This is a fixed-width field as previously described.

Packet Number Length (P): 2 bits. This is a fixed-width field as previously described.

Version ID (VID): 32 bits. This is a fixed-width field as previously described.

DCID Len (DLen): 1 byte; DLen <= 20. This is a fixed-width field, with a value constraint, as previously described.

Destination Connection ID (DCID): DLen bytes. This is a fixed-width field, with a length constraint, as previously described.

SCID Len (SLen): 1 byte; SLen <= 20. This is a fixed-width field, with a value constraint, as previously described.

Source Connection ID (SCID): SLen bytes. This is a variable-width field as previously described.

The syntax for extending sub-structures can be illustrated with the QUIC Retry Packet format [QUIC-TRANSPORT]. A Retry Packet is formatted as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               :
:                          Long Header                          :
:                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Retry Token                        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                     Retry Integrity Tag                       +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

where:

Long Header (LH): 1 Long Header; LH.T == 3. This field is a previously defined sub-structure. Its constraints can access fields in that sub-structure. In this example, the T field of the Long Header must be equal to 3.

Retry Token. This is a variable-length field as previously defined.

Retry Integrity Tag: 128 bits. This is a fixed-width field as previously defined.

As shown, the Long Header packet sub-structure is included. The Retry Packet enforces a new value constraint on the Long Packet Type (T) field.

3.6. Storing Data for Parsing

The parsing process may require data from previously parsed structures. This means that data needs to be stored persistently throughout the process. This data needs to be identified.

That the value of a particular field be stored upon parsing is indicated by the exact phrase "On receipt, the value of <field name> is stored as <stored name>." being present at the end of the description of a field.

An Initial Packet is formatted as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               :
:                          Long Header                          :
:                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

where:

Long Header (LH): 1 Long Header; LH.T == 0. This is field is a sub-structure, with a constraint, as previously defined. On receipt, the value of LH.DCID is stored as Initial DCID.

In this example, the value of the DCID field of the Long Header sub-structure is stored as Initial DCID.

3.7. Connecting Structures with Functions

The parsing or serialisation of some binary formats cannot be fully described without the use of functions. These functions take arguments (values from another structure), perform some computation, and generate a new structure.

Given the goal of fully capturing the parsing or serialisation of binary protocols, it is necessary to include the signature of these helper functions.

Function signatures are constructed by the word "func", followed by a space, then the name of the function. This is immediately followed by a set of brackets containing a comma separated list of the function's parameters, formatted as "<parameter name>: <parameter type>". This is followed by "->" and the return type of the function, followed by a colon.

The body of the function is not captured, owing to the complexity of both capturing and translating arbitrary code. As a result, it can be described in whichever format is most suitable for the document and its readership.

Those values that are stored persistently, as defined in Section 3.6, are accessible by functions.

As an example, the "apply_protection" function is defined as:

```
func apply_protection(to: Unprotected Packet)
                -> Protected Packet:
   apply packet protection to payload
   apply header protection to first_byte and packet_number
   construct appropriate Protected Packet based on first_byte
   return Protected Packet
```

In this example, 'Unprotected Packet' and 'Protected Packet' are existing types. The text in the function body is not interpreted.

A common use of functions is to specify operations used during packet parsing and serialisation.

To indicate that a PDU is created from another by way of a function, the sentence "A/An <PDU name A> is parsed from a <PDU name B> using the <function name> function" is used. This indicates that a PDU A is generated by passing PDU B into the named function. The function must take a single parameter, of the same type as PDU B, and return a PDU B.

To indicate that a PDU can be serialised to another by way of a function, the sentence "A/An <PDU name A> is serialised to a <PDU name B> using the <function name> function" is used. This indicates that a PDU B is generated by passing PDU A into the named function. The function must take a single parameter, of the same type as PDU A, and return a PDU B.

3.8. Specifying Protocols

A document will set out different structures that are not, on their own, protocol data units. To capture the parsing or serialisation of a protocol, it is necessary to be able to identify or construct those packets that are valid PDUs. As a result, it is necessary for the document to identify those structures that are PDUs.

The PDUs that comprise a protocol are identified using the exact phrase "This document describes the <protocol name> protocol. The <protocol name> protocol uses <list of PDU names>" where the list of PDU names is a comma separated list (with the last element, if there is more than one element, preceded by 'and'), each optionally preceded by "a" or "an". As a short-form, the variation "This document describes the <protocol name>, which uses <list of PDU names>" can be used as an alternative. The PDU names must be structure names defined in the document or a linked document. The PDU names are pluralised in the list. A document must contain exactly one instance of this phrase.

This document describes the Example protocol. The Example protocol uses Long Headers, STUN Message Types, and TCP Headers.

3.9. Importing Definitions from Other Documents

Protocols are often specified across multiple documents, either because the specification of a protocol's data units has changed over time, or because of explicit extension points contained in the protocol's original specification. To allow a document to make use of a previous PDU definition, it is possible to import PDU definitions (written in the format described in this document) from other documents.

A PDU definition is imported using the exact phrase "A/An ________ is formatted as described in <document identifier>". The document identifier must refer, unambiguously, to an existing document. An Internet-Draft is identified by its name. RFCs are identified by "RFC" followed by their number.

4. Design Principles

The use of structures that are designed to support machine readability might potentially interfere with the existing ways in which protocol specifications are used and authored. To the extent that these existing uses are more important than machine readability, such interference must be minimised.

In this section, the broad design principles that underpin the format described by this document are given. However, these principles apply more generally to any approach that introduces structured and formal languages into standards documents.

It should be noted that these are design principles: they expose the trade-offs that are inherent within any given approach. Violating these principles is sometimes necessary and beneficial, and this document sets out the potential consequences of doing so.

The central tenet that underpins these design principles is a recognition that the IETF standardisation process is not broken, and so does not need to be fixed. Failure to recognise this will likely lead to approaches that are incompatible with the standards process, or that will see limited adoption. However, the standards process can be improved with appropriate approaches, as guided by the following broad design principles:

Most readers are human: Primarily, standards documents should be written for people, who require text and diagrams that they can understand. Structures that cannot be easily parsed by people should be avoided, and if included, should be clearly delineated from human-readable content.

Any approach that shifts this balance -- that is, that primarily targets machine readers -- is likely to be disruptive to the standardisation process, which relies upon discussion centered around documents written in prose.

Writing tools are diverse: Standards document writing is a distributed process that involves a diverse set of tools and workflows. The introduction of machine-readable structures into specifications should not require that specific tools are used to produce standards documents, to ensure that disruption to existing workflows is minimised. This does not preclude the development of optional, supplementary tools that aid in the authoring machine-readable structures.

The immediate impact of requiring specific tooling is that adoption is likely to be limited. A long-term impact might be that authors whose workflows are incompatible might be alienated from the process.

Canonical specifications: As far as possible, machine-readable structures should not replicate the human readable specification of the protocol within the same document. Machine-readable structures should form part of a canonical specification of the protocol. Adding supplementary machine-readable structures, in parallel to the existing human readable text, is undesirable because it creates the potential for inconsistency.

As an example, program code that describes how a protocol data unit can be parsed might be provided as an appendix within a standards document. This code would provide a specification of the protocol that is separate to the prose description in the main body of the document. This has the undesirable effect of introducing the potential for the program code to specify behaviour that the prose-based specification does not, and vice-versa.

Expressiveness: Any approach should be expressive enough to capture the syntax and parsing process for the majority of binary protocols. If a given language is not sufficiently expressive, then adoption is likely to be limited. At the limits of what can be expressed by the language, authors are likely to revert to defining the protocol in prose: this undermines the broad goal of using structured and formal languages. Equally, though, understandable specifications and ease of use are critical for adoption. A tool that is simple to use and addresses the most common use cases might be preferred to a complex tool that addresses all use cases.

It may be desirable to restrict expressiveness, however, to guarantee intrinsic safety, security, and computability properties of both the generated parser code for the protocol, and the parser of the description language itself. In much the same way as the language-theoretic security ([LANGSEC]) community advocates for programming language design to be informed by the desired properties of the parsers for those languages, protocol designers should be aware of the implications of their design choices. The expressiveness of the protocol description languages that they use to define their protocols can force such awareness.

Broadly, those languages that have grammars which are more expressive tend to have parsers that are more complex and less safe. As a result, while considering the other goals described in this document, protocol description languages should attempt to be minimally expressive, and either restrict protocol designs to those for which safe and secure parsers can be generated, or as a minimum, ensure that protocol designers are aware of the boundaries their designs cross, in terms of computability and decidability [SASSAMAN].

Minimise required change: Any approach should require as few changes as possible to the way that documents are formatted, authored, and published. Forcing adoption of a particular structured or formal language is incompatible with the IETF's standardisation process: there are very few components of standards documents that are non-optional.

5. IANA Considerations

This document contains no actions for IANA.

6. Security Considerations

Poorly implemented parsers are a frequent source of security vulnerabilities in protocol implementations. Structuring the description of a protocol data unit so that a parser can be automatically derived from the specification can reduce the likelihood of vulnerable implementations.

As described in Section 4, the expressiveness of a protocol description language has implications for the safety, security, and computability properties of the parser for the protocol description language itself, and on the generated parser code for the protocols described using it. The language-theoretic security ([LANGSEC]) community explores the security implications of programming language design; the principles developed in that community should guide the development of protocol description languages.

7. Acknowledgements

The authors would like to thank Marc Petit-Huguenin for extensive feedback on the draft, including work on formalising the constraint syntax as given in Appendix A.1.

Wesley Eddy provided valuable feedback on the description format through adopting it in [RFC9293].

The authors would like to thank David Southgate for preparing a prototype implementation of some of the ideas described here.

This work has received funding from the UK Engineering and Physical Sciences Research Council under grant EP/R04144X/1.

8. Informative References

[RFC8357] Deering, S. and R. Hinden, "Generalized UDP Source Port for DHCP Relay", RFC 8357, March 2018, <https://www.rfc-editor.org/info/rfc8357>.

[QUIC-TRANSPORT] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-27, 21 February 2020, <https://www.ietf.org/archive/id/draft-ietf-quic-transport-27.txt>.

[RFC9000] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, May 2021, <https://www.rfc-editor.org/info/rfc9000>.

[RFC6958] Clark, A., Zhang, S., Zhao, J., and Q. Wu, "RTP Control Protocol (RTCP) Extended Report (XR) Block for Burst/Gap

Loss Metric Reporting", RFC 6958, May 2013, <https://www.rfc-editor.org/info/rfc6958>.

[RFC7950] Bjorklund, M., "The YANG 1.1 Data Modeling Language", RFC 7950, August 2016, <https://www.rfc-editor.org/info/rfc7950>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[ASN1] ITU-T, "ITU-T Recommendation X.680, X.681, X.682, and X.683", ITU-T Recommendation X.680, X.681, X.682, and X.683.

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

[RFC3550] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", RFC 3550, July 2003, <https://www.rfc-editor.org/info/rfc3550>.

[RFC8489] Petit-Huguenin, M., Salgueiro, G., Rosenberg, J., Wing, D., Mahy, R., and P. Matthews, "Session Traversal Utilities for NAT (STUN)", RFC 8489, February 2020, <https://www.rfc-editor.org/info/rfc8489>.

[RFC793] Postel, J., "Transmission Control Protocol", RFC 793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[LANGSEC] LANGSEC, "LANGSEC: Language-theoretic Security", <http://langsec.org>.

[SASSAMAN] Sassaman, L., Patterson, M. L., Bratus, S., and A. Shubina, "The Halting Problems of Network Stack Insecurity", ;login: -- December 2011, Volume 36, Number 6, <https://www.usenix.org/publications/login/

december-2011-volume-36-number-6/halting-problems-network-stack-insecurity>.

[RFC9293] Eddy, W., "Transmission Control Protocol", RFC 9293, August 2022, <https://www.rfc-editor.org/info/rfc9293>.

[NETWORKING2021] McQuistin, S., Band, V., Jacob, D., and C. Perkins, "Investigating Automatic Code Generation for Network Packet Parsing", Proceedings of the 20th International IFIP TC6 Networking Conference, Networking 2021, Espoo, Finland, June 21-24, 2021, <http://dl.ifip.org/db/conf/networking/networking2021/1570702659.pdf>.

Appendix A. ABNF specification

A.1. Constraint Expressions

```
constant = %x31-39 *(%x30-39)  ; natural numbers without leading 0s
short-name = ALPHA *(ALPHA / DIGIT / "-" / "_")
name = short-name *(" " short-name)
sp = [" "] ; optional space in expression
bool-expr = "(" sp bool-expr sp ")" /
           "!" sp bool-expr /
           bool-expr sp bool-op sp bool-expr /
           bool-expr sp "?" sp expr sp ":" sp expr /
           expr sp cmp-op sp expr
bool-op = "&&" / "||"
cmp-op = "==" / "!=" / "<" / "<=" / ">" / ">="
expr = "(" sp expr sp ")" /
      expr sp op sp expr /
      bool-expr "?" expr ":" expr /
      name / short-name "." short-name /
      "size(" short-name ")" /
      constant
op = "+" / "-" / "*" / "/" / "%" / "^"
length = expr sp unit / "[" sp name sp "]" / "variable length"
unit = %s"bit" / %s"bits" / %s"byte" / %s"bytes" / name
```

A.2. Augmented packet diagrams

Future revisions of this draft will include an ABNF specification for the augmented packet diagram format described in Section 3. Such a specification is omitted from this draft given that the format is likely to change as its syntax is developed. Given the visual nature of the format, it is more appropriate for discussion to focus on the examples given in Section 3.

Appendix B. Tooling & source code

The source for this draft is available from https://github.com/glasgow-ipl/draft-mcquistin-augmented-ascii-diagrams.

The source code for tooling that can be used to parse this document is available from https://github.com/glasgow-ipl/ips-protodesc-code. This tooling supports the automatic generation of Rust parser code from protocol descriptions written in the Augmented Packet Header Diagram format. It also provides test harnesses that demonstrate that the description of TCP [RFC9293] written in this format can be used to generate parser code.

Authors' Addresses

Stephen McQuistin
University of St Andrews
School of Computer Science
St Andrews
KY16 9AJ
United Kingdom

Email: sm@smcquistin.uk

Vivian Band
University of Glasgow
School of Computing Science
Glasgow
G12 8QQ
United Kingdom

Email: vivianband0@gmail.com

Dejice Jacob
University of Glasgow
School of Computing Science
Glasgow
G12 8QQ
United Kingdom

Email: dejice.jacob@glasgow.ac.uk

Colin Perkins
University of Glasgow
School of Computing Science
Glasgow
G12 8QQ
United Kingdom

Email: csp@csperkins.org