The Postcard Specification

Revision 1.x

OneVariable GmbH
Berlin, Germany

1 - Introduction

todo write an introduction

2 - TODO

Common stuff for all parts of the spec go here.

The Postcard Wire Format

Revision 1.x
OneVariable GmbH
Berlin, Germany

The Postcard Wire Format

version v1.x – 202x-yy-zz

Postcard is responsible for translating between items that exist as part of The Serde Data Model into a binary representation.

This is commonly referred to as **Serialization**, or converting from Serde Data Model elements to a binary representation; or **Deserialization**, or converting from a binary representation to Serde Data Model elements.

1 - Values

1.1 - Stability

The Postcard wire format is considered stable as of v1.0.0 and above of Postcard. Breaking changes to the wire format would be considered a breaking change to the library, and would necessitate the library being revised to v2.0.0, along with a new version of this wire format specification addressing the v2.0 wire format.

1.2 - Non Self-Describing

Postcard is NOT considered a "Self Describing Format", meaning that users (Serializers and Deserializers) of postcard data are expected to have a mutual understanding of the encoded data.

In practice this requires all systems sending or receiving postcard encoded data share a common schema, often as a common Rust data-type library.

Backwards/forwards compatibility between revisions of a postcard schema are considered outside of the scope of the postcard wire format, and must be considered by the end users, if compatible revisions to an agreed-upon schema are necessary.

Postcard may be extended to address some aspects expected in self-describing formats. See Appendix A - Postcard-RPC, for an example of a protocol that does so.

2 - `varint` encoded integers

For reasons of portability and compactness, many integers are encoded into a variable length format, commonly known as "leb" or "varint" encoded.

For the remainder of this document, these variable length encoded values will be referred to as `varint(N)`, where N represents the encoded Serde Data Model type, such as `u16` (`varint(u16)`) or `i32` (`varint(i32)`).

Conceptually, all `varint(N)` types encode data in a similar way when considering a stream of bytes:

1. The most significant bit of each stream byte is used as a "continuation" flag
2. If the flag is `1`, the this byte is NOT the last byte that comprises this varint
3. If the flag is `0`, then this byte IS the last byte that comprises this varint

All `varint(N)` types are encoded in "little endian" order, meaning that the first byte will contain the least significant seven data bits.

Data Type	Wire Type
`u16`	`varint(u16)`
`i16`	`varint(i16)`
`u32`	`varint(u32)`
`i32`	`varint(i32)`
`u64`	`varint(u64)`
`i64`	`varint(i64)`
`u128`	`varint(u128)`
`i128`	`varint(i128)`

Table A: Serde Integer `varint` Forms

As `u8` and `i8` types always fit into a single byte, they are encoded as-is rather than encoded using a varint.

Additionally the following two types are not part of the Serde Data Model, but are used within the context of postcard:

Data Type	Wire Type
`usize`	`varint(usize)`
`isize`	`varint(isize)`

Table B: Non-Serde Integer `varint` Forms

See the section “isize and usize” below for more details on these types are used.

2.1 - Unsigned Integer Encoding

For example, the following 16-bit unsigned numbers would be encoded as follows:

Dec	Hex	`varint` hex
0	`00_00`	`00`
127	`00_7F`	`7F`
128	`00_80`	`80, 01`
16383	`3F_FF`	`FF, 7F`
16384	`40_00`	`80, 80, 01`
16385	`40_01`	`81, 80, 01`
65535	`FF_FF`	`FF, FF, 03`

Table C: Unsigned Integer Examples

2.2 - Signed Integer Encoding

Signed integers are typically “natively” encoded using a Two’s Complement form, meaning that the most significant bit is used to offset the value by a large negative shift. If this form was used directly for encoding signed integer values, it would have the negative effect that negative values would ALWAYS take the maximum encoded length to store on the wire.

For this reason, signed integers, when encoded as a `varint`, are first Zigzag encoded. Zigzag encoding stores the sign bit in the LEAST significant bit of the integer, rather than the MOST significant bit.

This means that signed integers of low absolute magnitude (e.g. 1, −1) can be encoded using a much smaller space.

For example, the following 16-bit signed numbers would be encoded as follows:

Dec	Hex[1]	Zigzag	`varint` hex
0	`00_00`	`00_00`	`00`
−1	`FF_FF`	`00_01`	`01`
1	`00_01`	`00_02`	`02`
63	`00_3F`	`00_7E`	`7E`
−64	`FF_C0`	`00_7F`	`7F`
64	`00_40`	`00_80`	`80, 01`
−65	`FF_BF`	`00_81`	`81, 01`
32767	`7F_FF`	`FF_FE`	`FE, FF, 03`
−32768	`80_00`	`FF_FF`	`FF, FF, 03`

Table D: Signed Integer Examples

2.3 - Maximum Encoded Length

As the values that an integer type (e.g. `u16`, `u32`) are limited to the expressible range of the type, the maximum encoded length of these types are knowable ahead of time.

Postcard uses this information to limit the number of bytes it will process when decoding a `varint`.

As `varint`s encode seven data bits for every encoded byte, the maximum encoded length can be stated as follows:

```
bits_per_byte = 8
enc_bits_per_byte = 7
encoded_max = ceil(
  (len_bytes * bits_per_byte)
  / enc_bits_per_byte
)
```

Figure A: Max Encoded Size Pseudocode

[1]This column is represented as a sixteen bit, two’s complement form

Type	Varint Type	Type length (bytes)	Varint length max (bytes)
`u16`	`varint(u16)`	2	3
`i16`	`varint(i16)`	2	3
`u32`	`varint(u32)`	4	5
`i32`	`varint(i32)`	4	5
`u64`	`varint(u64)`	8	10
`i64`	`varint(i64)`	8	10
`u128`	`varint(u128)`	16	19
`i128`	`varint(i128)`	16	19

Table E: Maximum Encoded Lengths

2.4 - Canonicalization

The postcard wire format does NOT enforce canonicalization, however values are still required to fit within the Maximum Encoded Length of the data type, and to contain no data that exceeds the maximum value of the integer type.

In this context, an encoded form would be considered canonical if it is encoded with no excess encoding bytes necessary to encode the value, and with the excess encoding bits all containing `0`s.

Value	Encoded Form	Canon?	Valid?
`0`	`00`	Yes	Yes
`0`	`80 00`	No[2]	Yes
`0`	`80 80 00`	No[2]	Yes
`0`	`80 80 80 00`	No[2]	No[3]
`65535`	`FF FF 03`	Yes	Yes
`131071`	`FF FF 07`	No[4]	No[4]
`65535`	`FF FF 83 00`	No[2]	No[3]

Table F: Canonical Examples of `u16`s

2.5 - `isize` and `usize`

The Serde Data Model does not address platform-specific sized integers, and instead supports them by mapping to an integer type matching the platform's bit width.

[2] Contains excess encoding bytes
[3] Exceeds the Maximum Encoding Length
[4] Exceeds the maximum value of the encoded type

For example, on a platform with 32-bit pointers, `usize` will map to `u32`, and `isize` will map to `i32`. On a platform with 64-bit pointers, `usize` will map to `u64`, and `isize` will map to `i64`.

As these types are all `varint` encoded on the wire, two platforms of dissimilar pointer-widths will be able to interoperate without compatibility problems, as long as the value encoded in these types do not exceed the maximum encodable value of the smaller platform. If this occurs, for example sending `0x1_0000_0000usize` from a 64-bit target (as a `u64`), when decoding on a 32-bit platform, the value will fail to decode, as it exceeds the maximum value of a `usize` (as a `u32`).

3 - Variable Quantities

Several Serde Data Model types, such as `seq` and `string` contain a variable quantity of data elements.

Variable quantities are prefixed by a `varint(usize)`, encoding the count of subsequent data elements, followed by the encoded data elements.

4 - Encoding of Serde Data Model Types

4.1 - Primitives

"Primitive" types are data model types that always have the same encoding and form, and do not have names or types selected by the user.

4.1.1 - bool

A bool is stored as a single byte, with the value of 0x00 for false, and 0x01 as true.

All other values are considered an error.

4.1.2 - i8

An i8 is stored as a single byte, in two's complement form.

All values are considered valid.

4.1.3 - i16

An i16 is stored as a `varint(i16)`.

4.1.4 - i32

An i32 is stored as a `varint(i32)`.

4.1.5 - i64

An i64 is stored as a `varint(i64)`.

4.1.6 - i128

An i128 is stored as a `varint(i128)`.

4.1.7 - u8

An u8 is stored as a single byte.

All values are considered valid.

4.1.8 - u16

A u16 is stored as a `varint(u16)`.

4.1.9 - u32

A u32 is stored as a `varint(u32)`.

4.1.10 - u64

A u64 is stored as a `varint(u64)`.

4.1.11 - u128

A u128 is stored as a `varint(u128)`.

4.1.12 - f32

An `f32` will be bitwise converted into a `u32`, and encoded as a little-endian array of four bytes.

For example, the float value `-32.005859375f32` would be bitwise represented as `0xc200_0600u32`, and encoded as `[0x00, 0x06, 0x00, 0xc2]`.

> **NOTE:** f32 values are NOT converted to varint form, and are always encoded as four bytes on the wire. .

4.1.13 - f64

An f64 will be bitwise converted into a u64, and encoded as a little-endian array of eight bytes.

For example, the float value −32.005859375f64 would be bitwise represented as 0xc040_00c0_0000_0000u64, and encoded as [0x00, 0x00, 0x00, 0x00, 0xc0, 0x00, 0x40, 0xc0].

> **NOTE:** f64 values are NOT converted to `varint` form, and are always encoded as eight bytes on the wire. .

4.1.14 - `char`

A `char` will be encoded in UTF-8 form, and encoded as a string.

> **NOTE:** This encoding form is sub-optimal, and is likely to change in the next major revision to the Postcard Wire Format.
>
> Consider using `u32` (which will be `varint` encoded) for a single `char`, or use `string` rather than `seq(char)` for multiple `char`s.
>
> See issue postcard#101 for more details. .

4.1.15 - `string`

A `string` is encoded with a `varint(usize)` containing the length, followed by the array of bytes, each encoded as a single u8.

4.1.16 - `byte array`

A `byte array` is encoded with a `varint(usize)` containing the length, followed by the array of bytes, each encoded as a single u8.

4.1.17 - `unit`

The unit type is NOT encoded to the wire, meaning that it occupies zero bytes.

4.2 - Composite Types

Composite types are Data Model Types that have names or types selected by the user. They may also contain a variable number of child items, depending on the schema selected by the user.

4.2.1 - `option`

An option is encoded in one of two ways, depending on its value.

If an option has the value of None, it is encoded as the single byte `0x00`, with no following data.

If an option has the value of Some, it is encoded as the single byte `0x01`, followed by exactly one encoded Serde Data Type.

4.2.2 - `unit_struct`

The `unit_struct` type is NOT encoded to the wire, meaning that it occupies zero bytes.

4.2.3 - `newtype_struct`

A `newtype_struct` is encoded as the Serde Data Type it contains, with no additional data preceding or following it.

4.2.4 - `seq`

A `seq` is encoded with a `varint(usize)` containing the number of elements of the seq, followed by the array of elements, each encoded as an individual Serde Data Type.

4.2.5 - `tuple`

A `tuple` is encoded as the elements that comprise it, in their order of definition (left to right).

As `tuple`s have a known size, their length is not encoded on the wire.

4.2.6 - `tuple_struct`

A `tuple_struct` is encoded as a tuple consisting of the elements contained by the `tuple_struct`.

4.2.7 - map

A map is encoded with a `varint(usize)` containing the number of (key, value) elements of the map, followed by the array of (key, value) pairs, each encoded as a tuple of (key, value).

4.2.8 - `struct`

A `struct` is encoded as the elements that comprise it, in their order of definition (top to bottom).

As `struct`s have a known number of elements with known names, their length and field name

4.2.9 - `enum`

An `enum`, or “Tagged Union”, contains a variable number of Tagged Union Variants, depending on the schema of the type.

Tagged unions consist of two parts: The tag, or discriminant, and the value matching with that discriminant.

Tagged unions in postcard are encoded as a `varint(u32)` containing the discriminant, followed by the encoded value matching that discriminant.

The discriminants of an `enum` are numbered in the order of the definition of the variants (top to bottom), starting from `0`.

`enum`s do not appear on the wire, instead, exactly one of their variants will be encoded.

4.3 - Tagged Union Variants

4.3.1 - `unit_variant`

A `unit_variant` is an instance of a Tagged Union, consisting of a varint(u32) discriminant, with no additional encoded data.

4.3.2 - `newtype_variant`

A `newtype_variant` is an instance of a Tagged Union, consisting of a varint(u32) discriminant, followed by the encoded representation of the Serde Data Type it contains.

4.3.3 - tuple_variant

A `tuple_variant` is an instance of a Tagged Union, consisting of a `varint(u32)` discriminant, followed by a tuple consisting of the elements contained by the `tuple_variant`.

4.3.4 - `struct_variant`

A `struct_variant` is an instance of a Tagged Union, consisting of a varint(u32) discriminant, followed by a struct consisting of the elements contained by the `struct_variant`.

Postcard Schema Keys

James Munns
Revision 1.x
2025-0x-0y

Appendix A: The Postcard-Schema Key Calculation

version v0.x – 202x-yy-zz

The Postcard-Schema **Key** is a deterministic hash that can be used to identify messages.

As the Postcard Wire format is not self-describing, it is useful to have a method to identify the **kind** of messages when sent over the wire. This is intended to reduce cases where the sender and receiver of a message unexpectedly disagree on the expected message format, and to allow the receiver to reject messages it does not understand.

1 - Values

We gotta be small

We want to resist accidental changes

We don't claim resistance to malicious events

2 - `fnv1a64` hashing

Postcard-Schema Keys use the Fowler-Noll-Vo, or FNV non-cryptographic hash function.

As a hash algorithm, it was selected as it is simple to implement, and has reasonable avalanche characteristics, meaning that small changes to the input lead to large changes on the output.

Postcard-Schema keys specifically use the FNV-1a variant, which roughly follows the following pseudocode:

```
fn fnv1a(data: &[u8]) -> u64 {
  let mut hash = 0xcbf2_9ce4_8422_2325u64;

  for b in data {
    let ext = u64::from(*b);
    hash ^= ext;
    hash = hash.wrapping_mul(
      0x0000_0100_0000_01b3u64
    );
  }

  hash
}
```

When hashing multiple pieces of data separately, the data is treated "as if" the data was a single slice.

The remainder of this document uses the notation `hash(DATA)` to denote the `fnv1a64` hashing of each byte of `DATA` as described above

This remainder of this document also uses the notation `hash(DATA_A) + hash(DATA_B)` to describe a hash that is performed on each byte of `DATA_A` followed by each byte of `DATA_B`.

Therefore in this notation:

```
hash([0x01, 0x02]) + hash([0x03, 0x04])
```

would result in the same resulting value as in the notation:

```
hash([0x01, 0x02, 0x03, 0x04])
```

The resulting value of the `fnv1a64` hash is a 64-bit unsigned integer.

3 - Hash Inputs

A Key is formulated in terms of two pieces of data:

1. A **Path**, which is a UTF-8 text string
2. The **Schema** of a given type, which describes how the type is encoded in the Postcard Wire Format

For a given type `T`, and a given path `PATH`, the Key is calculated in the form:

```
key = hash(PATH) + hash(T::SCHEMA)
```

The intent is that changes to EITHER of the Path or Schema will result in a substantially different Key value.

The Path value is used to differentiate between different semantic meanings of a given type, for example, an `f32` value may be used to represent temperature in degrees Celsius, or may be used to represent distance in meters.

If these two pieces of data are given separate Paths, for example `"temperature/celsius"` and `"distance/meters"`, the differing Key values could be used to discriminate between them.

3.1 - Path hashing

The Path string is hashed using the bytes that make up the **UTF-8 code point sequence** of the string.

The string:

```
"temperature/celsius"
```

Would be comprised as the following sequence of bytes:

```
74 65 6D 70 65 72 61 74
75 72 65 2F 63 65 6C 73
69 75 73
```

and would produce the hashed value:

```
0x0353_7C16_0D8F_175Au64
```

3.2 - Schema hashing

The hash of a given type's Schema is calculated recursively, based on the Data Model Type information.

Each Data Model Type is assigned a one byte prime number that is used as an input to the hash. These primes were randomly selected from a list of all primes less than 256.

Data Model Type	Prime	Data Model Type	Prime
`bool`	`0x11`	`i8`	`0xC5`
`u8`	`0x3D`	`i16`	`0x1D`
`i32`	`0x0D`	`i64`	`0x0B`
`i128`	`0x02`	`u16`	`0x83`
`u32`	`0xD3`	`u64`	`0x13`
`u128`	`0x8B`	`usize`	`0x6B`
`isize`	`0x11`	`f32`	`0xEF`
`f64`	`0x71`	`char`	`0xC1`
`string`	`0x25`	`bytearray`	`0x65`
`option`	`0x6D`	`unit`	`0x47`
`seq`	`0x03`	`tuple`	`0xA7`
`map`	`0x4F`	`unit struct`	`0xBF`
`newtype struct`	`0x9D`	`tuple struct`	`0x05`
`struct`	`0x7F`	`enum`	`0xE9`
`schema`	`0xE5`	`unit variant`	`0xB5`
`newtype variant`	`0xDF`	`tuple variant`	`0xC7`
`struct variant`	`0x67`	-	-

Table G: Data Model Type Primes

3.2.1 - Primitive Type Hashing

For **Primitive**, the hashing of the type is complete after hashing the single byte prime. For example:

```
hash(f64::SCHEMA)
```

Would hash the single value `0x71`, and would produce the hashed value:

```
0xAF63_EC4C_8602_07BCu64
```

3.2.2 - Composite Type Hashing

For composite types that contain user selected data types, the single byte prime is hashed, and then the containing data's schema is hashed.

Note that this process may be recursive, and will recurse until the process terminates by reaching a Primitive Type.

3.2.2.1 - `option`

An `option` type's schema hash is calculated as:

```
hash(0x6D) + hash(T::SCHEMA)
```

3.2.3 - `seq`

A `seq` type's schema hash is calculated as:

```
hash(0x03) + hash(T::SCHEMA)
```

3.2.4 - `tuple`

A `tuple` type's schema hash is calculated using each of the N types that make up the tuple. For example, a 3-tuple, `(A, B, C)`, would be calculated as:

```
hash(0xA7)
 + hash(A::SCHEMA)
 + hash(B::SCHEMA)
 + hash(C::SCHEMA)
```

3.2.5 - `map`

A `map` type's schema hash is calculated using the key type `K` and value type `V` that make up the map. For example, a `map<K, V>` would be calculated as:

```
hash(0x4F)
  + hash(K::SCHEMA)
  + hash(V::SCHEMA)
```

3.2.6 - `unit struct`

A `unit struct` is considered a primitve for the purposes of hashing, and is defined as

```
hash(0x9D)
```

3.2.7 - `newtype struct`

3.2.8 - `tuple struct`

3.2.9 - `struct`

3.2.10 - `enum`

4 - TODO

When encoded in the Postcard Wire Format, Postcard-Schema Keys are encoded as a tuple of eight bytes in little-endian order, rather than an a `varint(u64)`.

The Postcard-RPC Protocol

James Munns
Revision 1.x
2025-0x-0y

Appendix B: The Postcard-RPC protocol

version v0.x – 202x-yy-zz

Postcard-RPC is a point to point connection protocol. It connects a **client** and a **server**.

1 - Values

As a protocol, The Postcard-RPC Protocol is intended to transit across many kinds of transports, such as USB, Bluetooth, TCP, UART/Serial Ports, or any other method of conveying frames.

It aims to offer **just enough functionality** to make it useful, while still being misuse and accident resistant.

It is intended to be a lightweight protocol, suitable for communication with microcontroller devices. For this reason, there are many things it **does not do**, or **does not guarantee**, to prioritize simplicity of implementation.

2 - Major Concepts

The following sections are a progressive introduction into the aspects of the protocol.

2.1 - Frames

At the lowest level, the Postcard-RPC protocol is made up of **Frames**, which are a variable-sized container of bytes.

Postcard-RPC does **NOT** define how these frames are transported, and it is expected that they may be modified during transit: changing encoding, adding additional metadata or integrity checks, or adding of encryption. These aspects are expected to be defined by the **Wire Interface**, discussed later.

Frames consist of two main parts:

1. A **Header**, containing limited metadata about the frame in a fixed format
2. A **Body**, containing user-defined data in the postcard encoding format

2.1.1 - The Header

A **Header** contains three pieces of information:

1. A **Tag**, which encodes the version of the header and remaining content of the Frame
2. A **Key**, which is a hash of the schema of the Body and the Endpoint Name
3. A **Sequence Number**, which is number used for identifying the instance of the Frame.

The Key and Sequence Number are variable length fields. The length of these fields is determined by the contents of the Tag field.

2.1.1.1 - Header Tag

The Header Tag is always the first byte of the Frame. This byte is a bitfield containing three fields:

1. The **Version**, consisting of four bits
2. The **Key Length**, consisting of two bits
3. The **Sequence Number Length**, consisting of two bits.

The Header tag takes the following form:

```
.---------- MSBit
|        .-- LSBit
v        v
KK_SS_VVVV
^^ ^^ ^^^^-- Version
|  '-------- Sequence Number Length
'----------- Key Length
```

Figure B: Version field contents

The values of these bits control how the Header should be decoded and the length of the header. Any frame containing contents marked **INVALID** must be rejected.

Any frame shorter than the length reported by the header tag must be rejected.

Value	Sequence Number Length
`0b0000`	Version 1
-	INVALID

Table H: Version field contents

Value	Sequence Number Length
`0b00`	1 Byte
`0b01`	2 Bytes
`0b10`	4 Bytes
`0b11`	INVALID

Table I: Sequence Number Length field contents

Value	Key Length
`0b00`	1 Byte
`0b01`	2 Bytes
`0b10`	4 Bytes
`0b11`	8 Bytes

Table J: Key Length field contents

2.1.1.2 - The Key

The Key is encoded as a little-endian unsigned integer, of the length reported in the Tag.

The Key appears directly after the Tag.

The Key field is used to uniquely identify the contents of the Body, both for routing purposes, as well as detecting when the schema of the Body has changed.

2.1.1.3 - The Sequence Number

The Sequence Number is encoded as a little-endian unsigned integer, of the length reported in the Tag.

The Sequence Number appears directly after the Key.

The Sequence Number is used to uniquely identify messages, as well as in some cases, correlate between requests and responses.

2.1.2 - The Body

After the header, all remaining bytes in the Frame are considered part of the Body. The Body of the Frame may be any length of bytes, including zero. The length of the Body is determined and reported by the Wire Implementation.

The Body appears directly after the Sequence Number.

The Body is always encoded in the Postcard format. The type and schema of the Body is determined by the Client or Server using the Key field of the header.

2.2 - Roles

There are two roles in Postcard-RPC, the **Client** and the **Server**. Generally, the Client acts as the "initiator" of communications, sending Requests that are handled by the Server, and elicit a Response.

2.3 - Methods of Communication

There are two core methods of communication in Postcard-RPC: **Endpoints** and **Topics**.

2.3.1 - Endpoints

Endpoints are transactional operations, initiated by the Client, and defined by the Server. A Server may have any number of Endpoints.

Endpoints consist of a **Request**, sent by the Client, and a **Response**, sent by the Server. Both the Request and the Response are sent as a single Frame.

When sending a Request, the Client selects a Sequence Number. The Server will always use the same Sequence Number in the Response.

A Server will have a set of Endpoints that it supports, defined by three pieces of information:

1. The Schema of the Request Frame
2. The Schema of the Response Frame
3. A UTF-8 string which serves as the Name of the Endpoint

This information will be used to calculate two Keys:

1. The Key of the Request, calculated using the Schema of the Request Frame and the Name of the Endpoint
2. The Key of the Response, calculated using the Schema of the Response Frame and the Name of the Endpoint

TODO: Describe the specific calculation somewhere.

The lifecycle of an Endpoint communication is as follows:

2.3.1.1 - The Client sends a Request Frame

The client sends an outgoing Request:

1. The Client selects the length and value of the Sequence Number

2. The Client uses the Key of the Request
3. The Client fills the body with a message matching the Schema of the Request Frame

The client then begins listening for an appropriate Response Frame from the Server.

2.3.1.2 - The Server receives the Request

The Server uses the Request Key to dispatch to the appropriate handler for this Endpoint.

2.3.1.3 - The Server sends a Response

The Server will send exactly one of the following potential responses:

2.3.1.3.1 - On Success

If the Request was processed successfully:

1. The Server selects the same length and value of the Request Sequence Number
2. The Server selects the Key of the Response
3. The Server fills the body with a message matching the Schema of the Response Frame

2.3.1.3.2 - On Failure

If the Request Key was unknown, or an error occurred during processing, such as a failure to decode the Request frame, and the Request was NOT processed successfully:

1. The Server selects the same length and value of the Request Sequence Number
2. The Server select the Key of the Error
3. The Server fills the body with a message matching the Schema of the Error Frame

2.3.2 - The Client receives the Response

The client will receive the Response Frame sent by the server, and determine whether to decode this response as either the expected Response or as an Error, depending on the Key of the Response Frame.

2.3.3 - Topics

Topics are non-transactional operations. Unlike Endpoints, they may be initiated by EITHER the Client or the Server.

Topic messages are intended for use in situations where it is not reasonable to use Endpoints. This typically takes one of two forms:

1. Messages that happen often, and would be burdensome to poll for. For example: sensor data sent at a high polling rate, sending data every 5ms.
2. Messages that happen extremely rarely, and would be burdensome to poll for. For example: button press events that may happen hours apart.

When Topic messages are sent by the Client, they are considered **Topic-In** messages. When Topic messages are sent by the Server, they are considered **Topic-Out** messages.

Topic Messages are always a single Frame.

When sending a Topic Message, the sender selects a Sequence Number.

The Server and the Client will each have a set of Topics that they support sending or receiving, defined by three pieces of information:

1. The Schema of the Request Frame
2. The Schema of the Response Frame
3. A UTF-8 string which serves as the Name of the Endpoint

This information will be used to calculate the Key of the Message, calculated using the Schema of the Message Frame and the Name of the Topic.

TODO: Describe the specific calculation somewhere.

Topic Messages are sent without regard to whether the other party are interested or capable of processing the message. No response is sent, regardless of whether the