



Do Pure Ruby Dreams Encrypted Binary Protocol?

unasuke (Yusuke Nakamura)

RubyKaigi 2022
2022-09-09

Self introduction

- Name: unasuke (Yusuke Nakamura)
- Work: freelance Web app developer @ Tokyo
 - Rails app developer (mainly)
- Itamae gem maintainer, Kaigi on Rails Organizer
- GitHub <https://github.com/unasuke>
- Mastodon <https://mstdn.unasuke.com/@unasuke>
- Twitter https://twitter.com/yu_suke1994



RubyKaigi 2022

At first, do PureRuby Dream of Encrypted Binary Protocol?

A: There is a harsh reality. 



RubyKaigi 2022

A brief explanation of what QUIC is

- QUIC: standarized at 2021 by RFC 9000 etc
 - Using UDP for communication

TCP

low efficiency, high reliability

UDP

high efficiency, low reliability → faster than TCP



RubyKaigi 2022

Did you remember my last year talk about QUIC?

Ruby, Ractor, QUIC

unasuke (Yusuke Nakamura)

RubyKaigi Takeout 2021

2021-09-11



RubyKaigi Takeout 2021

<https://slide.rabbit-shocker.org/authors/unasuke/rubykaigi-takeout-2021/>



RubyKaigi 2022

Ractor, I dropped out

It's too difficult. Because...

- Implementing communication protorol is very hard work
- debugging code using Ractor is very hard in now
 - debug.gem, I hope it will be a savior in the future...

Try to solve two problems in one time is difficult.

→ I gave up to use Ractor in first implementation 😞



It's a binary protocol

message like that

```
{ "message": "Hello!", "kind": "greet" }
```

parse it by ruby

```
require "json"
request = JSON.parse('{ "message": "Hello!", "kind": "greet" }')
pp request["message"] # => "Hello!"
```



RubyKaigi 2022

It's a binary protocol

QUIC message

```
c000000001088394c8f03e5157080000449e7b9aec34d1b1c98dd7689fb8ec11  
d242b123dc9bd8bab936b47d92ec356c0bab7df5976d27cd449f63300099f399  
1c260ec4c60d17b31f8429157bb35a1282a643a8d2262cad67500cadb8e7378c  
8eb7539ec4d4905fed1bee1fc8aaafba17c750e2c7ace01e6005f80fcbb7df6212  
30c83711b39343fa028cea7f7fb5ff89eac2308249a02252155e2347b63d58c5  
457afd84d05dfffd20392844ae812154682e9cf012f9021a6f0be17ddd0c208  
4dce25ff9b06cde535d0f920a2db1bf362c23e596d11a4f5a6cf3948838a3aec  
4e15daf8500a6ef69ec4e3feb6b1d98e610ac8b7ec3faf6ad760b7bad1db4ba3  
...
```



It's a binary protocol

Reading QUIC message (same as last year's slide)

```
c00000001088394c8f03e515788888449e7b9aec3d1b1c88dd7689fb8ec11  
d242b123dc9bd8ba0936b47d92ec356cbab7df597ed27  
1c260ec4c68d17b31f8429157b35a1282a643a8d2262cad07500cad000000000000  
0eb7539ec4d4905fed1beefc8aaefba17c750e2c7ae01e6005f80fcbb7df6212  
30c83711b39343fe828ce777bf89eac2308249a02252155e2347b63d58e5  
457af8d485dffffdb28392844ae02154682e9cf81921a6ffab17d00000000000000  
4dc2e25f9bb6cd535dbf928a2db1bf362c23e96d11aaefca3f948838a3ae3  
ee15daf85080adef69ec4e3fbefb1d98e618ac8b7ac3faefad760b7bad1db4ba3  
4856ea94dc258aef3fdb41d15fb6a8e5eba0fc3dd06bc8e30c5c428753885db  
8599e864db2f64264ed5e395be2e20d82d1560da8d5d998cabade85389eae0c  
7b4376e846d29f37ed7b4ea9efc5082e7961b7f259a323851f081d582393aa5f8  
99375a67258bf63ad61fa8b1d96dhd4faddfcfc5266ba611722295c986556  
be52afe3f565636ad1b17d588b73d8743eeb524be22b3dbc2c7468d54119c74  
68449e13d8e3b95811a198f3491de3fe78e942b338487abf82a4ed7cb311663a  
c9899f4157815853d91e23837c227a33cd5ec281ca3f79c44540b9d98ca09  
fb864c99e3d097911d397e9c5db0238229a234cb38186c4819e889c5927726632  
291d6a418211cc29662e28fe7fe3edf338f2c680a9d8cfcb5699dbfe58964  
25c5bac4aee82e5785aaef4e2513eaf85796b87ba2ee47d8886f8d2c25e58fd  
14de11e6c418559382f939b6e1abd5761279c4b2e0fb85c1728ff18f5891ff  
ef152eeff2fa89546aee33c28eb158ff28fb57b7605533341l321999d20811a198  
e3fc435f9f254181aee17c1bf28258ff6847472f36857fe843919f59848899d  
c324844e847ad4fa8ab24ff719595d9e37252d6225265e9b8a39392061085349d73  
203a4a13e96f432ec8fd4aee65accdd5e3984df54c1da51b0ff28dc0c77f  
cb2c0ebeb085cb8584db87632cf3db84daee6785769d1de354270123cb11458e  
fc68a047683d7bd8df811365565fd98c4ceb936bcab8089fc33bd881b083de  
a2el1fc5aa63d8Rca19896d2bf59a071b851ebc239952172f296fb65e724847  
98a2181014f3b94a4e97d17b438130368cc39dbb2d198065a39865479326cd2  
162748a291fbc3c8745c0f50fb3a3852e566d44575c29d39a3fcd7219846f4  
408591f355e12d439ff150aab7613499dbd49adabc8076ee823b15b65fbfc5ca9  
6948189f23f3580db82123535eb8a7433bdabc909271aeeccb58939a8c4d4e  
872e6ff58800175f113253d8f7a9c48885c2f552e657d6c683f252e1a8e3887f76f0  
be79e2fb75f5dsfbbe2a38eacdd220t23cc8aae8878cdcb3862863ff8789499  
54da48781893a7e69adsff4af380cd8846b6279ph3ff3afbd64491c85194aab  
7608d8086654f9f4488eb3b28591356fbf6425aca2d05244259ff2b19c41b9  
f967sc9e9c1dde434da7d2d392b985d03d1f9a793d1fa5958b0d4937faa731b4  
856d731bd267b698aae7983aaef7598e8a39813137aae3d484f518cfda4884644  
7e78bf706ca4c75a9c5453e9f7cf72b84cd169a44e55c88d4a97f9474241  
e221a144868018ab8856972e194cd934
```

header (long header)

frame

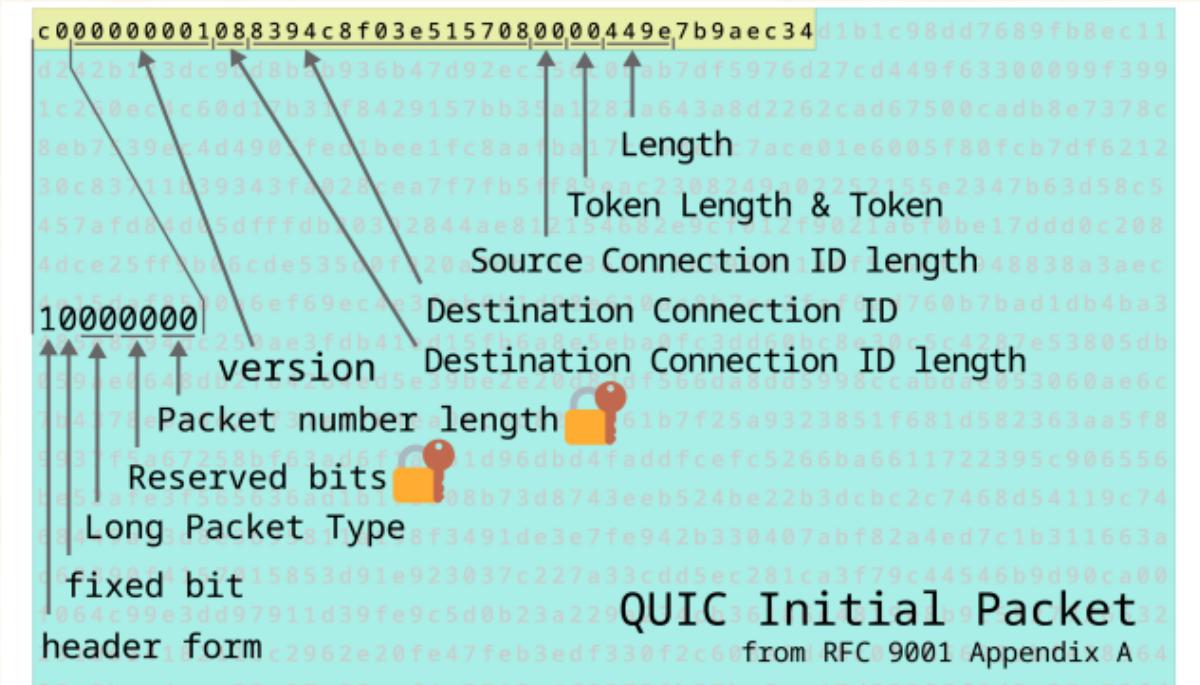
QUIC Initial Packet
from RFC 9001 Appendix A



RubyKaigi 2022

It's a binary protocol

Reading QUIC message (same as last year's slide)



It's a binary protocol

Reading QUIC message (same as last year's slide)

c 000000001088394c8f03e5157080000449e7b9aec34d1b1c98dd7689fb8ec11
d242b123dc9bd8bab936b47d92e0350c0ab7df5976d27cd49f63300099f399
1c260ec4c60d17b31f8429157ba35a128a643a8d2262cad6750cadb8e7378c
8eb7539ec4d4905fed1bee1f8aefba17c750e2c7ace01e10010dfcb7df6212
30c83711039343fa028cea77f05ff89eac2308249a02252155e2347b63d58c5
457af84d05dfffd20392844e812154682e9cf012f9021a6f0be17ddd0c208
4dce25f9b06cde535d0f920a2db1bf362c234pn_offset = 67+8+0+2+1+0 ec
4e15da18500a6ef69ec4e31eb6b1d98e610ac8b7a-3f9f6d769b7bad1db4ba2
485e8a94dc250aeefdb47ed15fb6a8e5ea0fc3d068c8e30c5c4287c53805db
059ae0648db2f04264e5e39beLength (2)
7b4378e846d20f37d47b4e9ec5d82e7961b7f25a9333851f681d582363aa5f8
9937f5a67258bf63ad0f1a00190dd4faddfcerc5260ba6611722395c906556
be52afSource Connection ID length b(0)be22b3dcbc2c7468d54119c74
6440134e3b15811a10ef344d3e77f9430407abf82a4ed7c1b311663a
Destination Connection ID length (8)c69890f4157015853d91e923037c227a33cdd5ec281ca3f79c44546b9d90ca00
f064c99e3dd97911d39fe9c5d0b23a229f03640409109from RFC 9001 Appendix 8A 64
291d6a418211cc2962e20fe47feb3edf330f2c66
RubyKaigi 2022

sample

sample_offset = pn_offset + 4

Token Length (1) & Token (0)

Source Connection ID length b(0)

Destination Connection ID length (8)

QUIC Initial Packet

from RFC 9001 Appendix 8A 64



It's a binary protocol

Reading QUIC message (same as last year's slide)

```
c000000001088394c8f03e515708000449e7b9aec34d1b1c98dd7689fb8ec11
d242b123dc9b d8bab936b47d92ec356c0bab7df5976d27cd 49f63300099f399
1c260ec4c60d17b31f8429157bb35a1282a643a8d2262cad67500cadb8e7378c
8eb7539ec4d4905fed1bee1fc8aaafba17c750e2c7ace01e6006f80fcbb7df6212
30-027344b30343f-028-~-773f6c5ff80-~-7388240-0225315f-02347b63d58c5
enc = OpenSSL::Cipher.new('aes-128-ecb')
enc.encrypt

# header protection key
enc.key = ["9f50449e04a0e810283a1e9933adedd2"].pack("H*")

mask = ""
mask << enc.update(sample)
mask << enc.final

mask = 437b9aec36
```

sample

QUIC Initial Packet

from RFC 9001 Appendix A 54



RubyKaigi 2022

It's a binary protocol

Reading QUIC message (same as last year's slide)

```
c00000001088394c8f03e5157080000449e7b9aec34d1b1c98dd7689fb8ec11  
d42b123dc9bd8bab936b47d92ec356c0babd5976d27cd449f63300099f399  
1c260ec4c60d17b31f8429157bb35a1282a543a8d2262cad67500cadb8e7378c  
8e67539ec4d4905fed1bee1fc8aa1c01c70c2c70e01e00000001c07df6212  
header[18..21] ^= mask[1..4]  
30083711b39343fa028cea7f7fb5 = 00000002249a02252155e2347b63d58c5  
4570fd84d05dfffd20392844ae812154682e9cf012f9021a6f0be17ddd0c208  
4dcce25ff9b06cde535d0f920a2db1bf362c23e596d11a4f5a6cf3948838a3aec  
4e154af8500a6ef69ac4e3feb6b1d98e610ac8b7ec3faf6ad760b7bad1db4ba3  
485cc0948c250ac51ab1cd151b6a8e5eba0fc3dd60bc8e30c5c4287e53805db  
header[0] ^= mask[0] & 0x0f  
0= c3  
0648db2f64264ed5e39be2e20d82df566da8dd5998ccabdae053060ae6c  
7b4378e846d29f37ed7b4ea9ec5d82e7961b7f25a9323851f681d582363aa5f8  
9037f5a67258bf63ad5c1a0b1d004bd1fadd5ec6c5256b66c11722305c006656  
true header = c30000001088394c8f03e5157080000449e0000002  
beszate31563636audio17d50007308745eeb524be22b30c0c2c7408034119c74  
68449a13d8e3b95811a198f3491de3e7fe942b330407abf82a4ed7c1b311663a  
mask = 437b9aec36  
f064c99e3dd97911d39fe9c5d0b23a229  
QUIC Initial Packet  
291d6a418211cc2962e20fe47feb3edf330f2c60 from RFC 9001 Appendix A 64  
291d6a418211cc2962e20fe47feb3edf330f2c60 from RFC 9001 Appendix A 64
```



It's a binary protocol

Reading QUIC message (same as last year's slide)

```
c588888881888394c8f03e515788888449e00000002d1b1c98dd7689fb8ec11  
d242b123dc9bd8ba0936b47d92ec356cbab7df597d022cd449f633888997399  
1c260ec4c60d17b31f8429157b235a1282a643a8d2262cad67500cadb8e7378c  
0eb7539ec4d4905fedbeef1fcfaafba17c758e2c7ace01e6005f80fc7df6212  
38c85711b59343fe825cea77fbffff89eac238824902252159e2342b03d98c5  
457a7f8d4d85dfffd828392844a812154682e9cf012fn021a0f#be17d9d8c288  
4dc2e25f9bb6c  
4e15daf8500a6  
485e6a94dc258  
859ee0648db2f  
7b4378e846d29  
99375a67258b  
be52afe3f5650  
68449a13d8e3b  
c9899f415781  
f864c99e3d997  
291d6a418211c  
25c5bac4aee82  
14de71e6c4185  
ef152eeef2fa89  
e3fc43879f254  
c324844ae847a4  
203a4a13e96f5  
cb2c0ebeb085c  
fc68ac47683d7  
a2e1fb5aa63  
98a2181014f3b  
162748a29f8c3  
40591f355e12d439ff150aab7613499dbd49adabc8676eff923b15b85bfcc5a9  
69481b9f23f3580b82123535e8a7433bdabcb909271a9e6cbcb58930a88c4e  
872e6ff5880175f113253d8f8a9ca8885c2f552e657dg03f252e1aae308f769  
be79e2fb5f5dsfbba2e38ecadd220t723c8caeaa8879cdcb38682637ff8799480  
54da48781893a7e69adsff4af780cd884a6b0279ab3ff3a7b64491c85194aab  
70826a080654f9f4488eb28591356fbf0425aca20d8c5244259ff2b19c4199  
f967sc9e9c1dde434da7d2d92b9850df3d1f9a793d1fa7595b0d4937faa73194  
856d731bd267b698aa879831aaaf579b6fa39813137aaac6d484f518cfda488464  
7e78bfe706ca4c75a0c5453e9f7cf7b8b4c3d169a44e55ca88d4a97f9474241  
e221af44868018ab0856972e194cd934
```

QUIC Initial Packet
from RFC 9001 Appendix A



RubyKaigi 2022

It's a binary protocol

Convert to bit-by-bit representation

```
"Hello".unpack1("B*")  
# => "0100100001100101011011000110110001101111"
```



RubyKaigi 2022

It's a binary protocol

Oops!

```
data = "Hello".unpack1("B*")
# ...snip...
data.unpack1("B*") # unpack twice!
# => "001100000011000100110000001100000011000100110....
```

I wasted a lot of time because of this mistake. 



RubyKaigi 2022

How are the other language implementations

Look around some QUIC impletemtations

- kwik (Java)
- quic-go (Go)
- cloudflare/quiche (Rust)
- aioquic (Python)



RubyKaigi 2022

kwik (Java) : QUIC impletemtations

```
void parsePacketNumberAndPayload(ByteBuffer buffer, byte flags, int  
remainingLength, Keys serverSecrets, long largestPacketNumber, Logger  
log) throws DecryptionException, InvalidPacketException {
```

[https://github.com/ptrd/kwik/blob/d1c52e6ac3/src/main/java/net/luminis/quic/packet/
QuicPacket.java#L89](https://github.com/ptrd/kwik/blob/d1c52e6ac3/src/main/java/net/luminis/quic/packet/QuicPacket.java#L89)



RubyKaigi 2022

quic-go (Go) : QUIC impletemtations

```
type unpackedPacket struct {
    packetNumber    protocol.PacketNumber // the decoded packet number
    hdr            *wire.ExtendedHeader
    encryptionLevel protocol.EncryptionLevel
    data           []byte
}
```

[https://github.com/lucas-clemente/quic-go/blob/66f6fe0b711bc/
packet_unpacker.go#L29-L34](https://github.com/lucas-clemente/quic-go/blob/66f6fe0b711bc/packet_unpacker.go#L29-L34)



RubyKaigi 2022

cloudflare/quiche (Rust) : QUIC impletemtations

```
impl<'a> OctetsMut<'a> {
    /// Creates an `OctetsMut` from the given slice, without copying.
    ///
    /// Since there's no copy, the input slice needs to be mutable to allow
    /// modifications.
    pub fn with_slice(buf: &'a mut [u8]) -> Self {
        OctetsMut { buf, off: 0 }
    }
}
```

<https://github.com/cloudflare/quiche/blob/3131c0d37/octets/src/lib.rs#L329-L336>



RubyKaigi 2022

aioquic (Python) : QUIC impletemtations

```
@dataclass
class QuicStreamFrame:
    data: bytes = b""
    fin: bool = False
    offset: int = 0
```

<https://github.com/aiortc/aioquic/blob/c758b4d936/src/aioquic/quic/packet.py#L477-L481>



RubyKaigi 2022

back to the Ruby

- Those languages have Byte specific class or types
 - but Ruby is not
 - but we can use String or array of Integer

I would like to see a binary protocol implementation by Ruby that already exists.

→ MessagePack!



msgpack/msgpack-ruby

MessagePack is an efficient binary serialization format.

<https://msgpack.org>



RubyKaigi 2022

msgpack/msgpack-ruby

```
void _msgpack_unpacker_init(msgpack_unpacker_t* uk)
{
    msgpack_buffer_init(UNPACKER_BUFFER_(uk));

    uk->head_byte = HEAD_BYTE_REQUIRED;

    uk->last_object = Qnil;
    uk->reading_raw = Qnil;

    uk->stack = _msgpack_unpacker_new_stack();
}
```

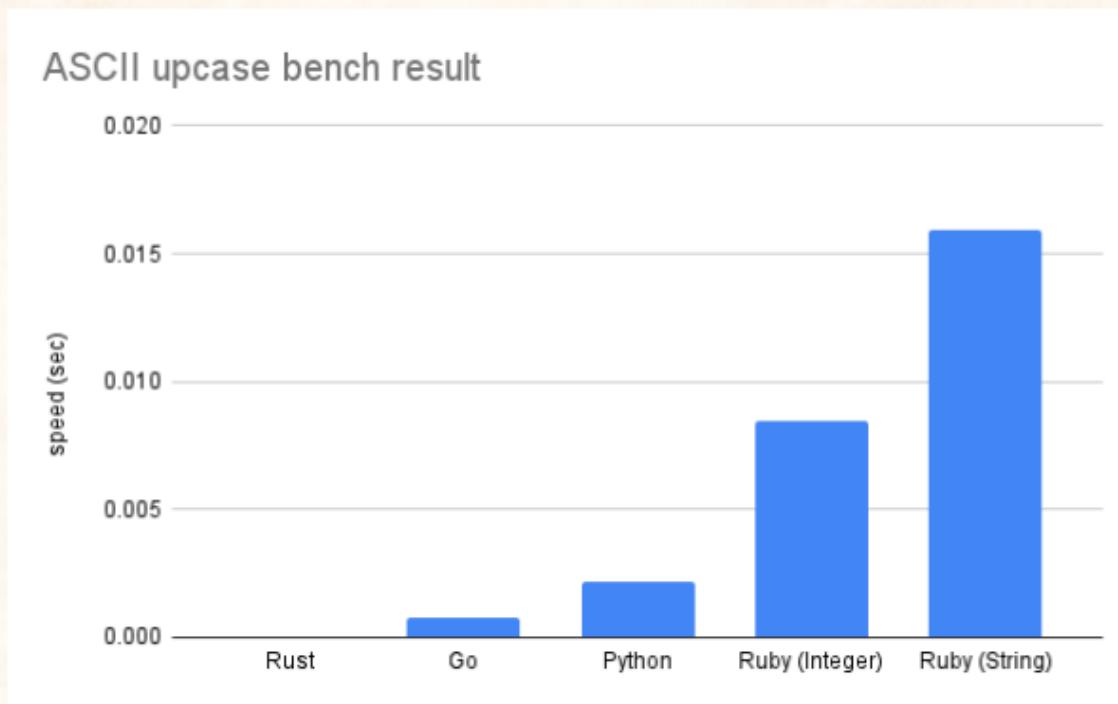
<https://github.com/msgpack/msgpack-ruby/blob/0775a9a6a5/ext/msgpack/unpacker.c#L75-L85>



RubyKaigi 2022

compare languages

benchmark for bytes manipulation (AWS c5.large Ubuntu 22.04)



RubyKaigi 2022

compare String and array of Integer

```
$ bundle exec ruby main.rb
Warming up -----
  hello_world_upcase_string    848.029k i/s -    872.655k times in 1.029039s (1.18µs/i)
  hello_world_upcase_integer   1.268M i/s -    1.291M times in 1.018188s (788.58ns/i)
Calculating -----
  hello_world_upcase_string    861.605k i/s -    2.544M times in 2.952730s (1.16µs/i)
  hello_world_upcase_integer   1.285M i/s -    3.804M times in 2.961585s (778.48ns/i)
```

Comparison:

```
hello_world_upcase_integer : 1284551.4 i/s
  hello_world_upcase_string : 861605.1 i/s - 1.49x slower
```



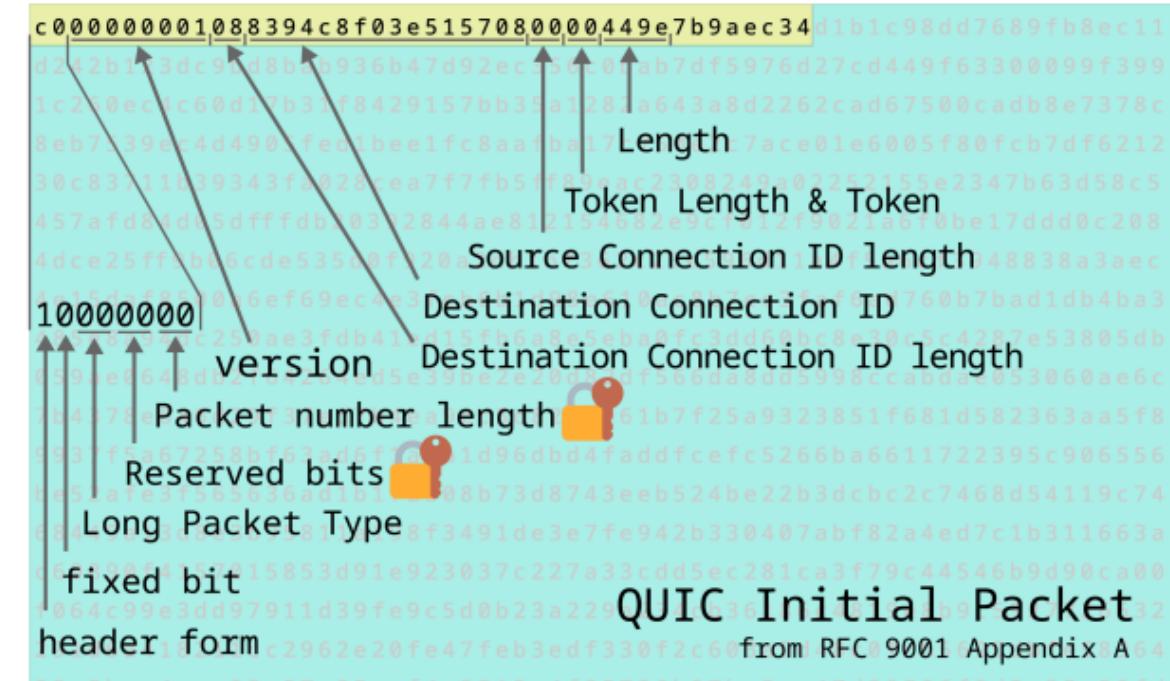
RubyKaigi 2022

How to prevent mistake?

```
class UnpackedString < String
  def unpack1()
    raise RuntimeError
  end
end
```



It's an encrypted protocol



It's an encrypted protocol

compare String and array of Integer (re)

```
$ bundle exec ruby main.rb
Warming up -----
hello_world_upcase_string    848.029k i/s -    872.655k times in 1.029039s (1.18µs/i)
hello_world_upcase_integer    1.268M i/s -    1.291M times in 1.018188s (788.58ns/i)
Calculating -----
hello_world_upcase_string    861.605k i/s -    2.544M times in 2.952730s (1.16µs/i)
hello_world_upcase_integer    1.285M i/s -    3.804M times in 2.961585s (778.48ns/i)
```

Comparison:

```
hello_world_upcase_integer : 1284551.4 i/s
hello_world_upcase_string  : 861605.1 i/s - 1.49x slower
```



RubyKaigi 2022

It's an encrypted protocol

leading "0" gone

```
"01001".to_i(16) # => 4097  
"01001".to_i(16).to_s(16) # => "1001"
```



RubyKaigi 2022

XOR between String

```
def xor(a, b)
  a.unpack("C*").zip(b.unpack("C*")).map do |x, y|
    x ^ y
  end.pack("C*")
end
```



It's an encrypted protocol

```
data[0] = [(data[0].unpack1('H*').to_i(16) ^
(mask[0].unpack1('H*').to_i(16) & 0x0f)).to_s(16)].pack("H*")  
  
# https://www.rfc-editor.org/rfc/rfc9001#figure-6
pn_length = (data[0].unpack1('H*').to_i(16) & 0x03) + 1  
  
packet_number =
(data[pn_offset...pn_offset+pn_length].unpack1("H*").to_i(16) ^
mask[1...1+pn_length].unpack1("H*").to_i(16)).to_s(16)  
  
# fill zero because leading "0" gone
data[pn_offset...pn_offset+pn_length] =
[("0" * (pn_length * 2 - packet_number.length)) + packet_number].pack("H*")
self.class.new(data, protected: false).tap{|p| p.parse}
```

XOR between byte (Python)

```
bytes([aa ^ bb for aa, bb in zip(a, b)])
```

<https://programming-idioms.org/idiom/238/xor-byte-arrays/4146/python>



RubyKaigi 2022

What should we do?

If there is no constraint for "Pure Ruby"...

- Write extension library by system programming language
 - C or Rust
 - Rust is popular
 - <https://github.com/rubygems/rubygems/pull/5613>

"Add support for bundle gem --rust command"



RubyKaigi 2022

QUIC, Some headers and many frames

- two type of headers
 - long header, short header
- 20 types of frames
 - padding, ping, ack, etc...



RubyKaigi 2022

QUIC, Some headers and many frames

```
private def find_frame_type(frame)
  case [frame[0..7]].pack("B*")
  when "\x00"
    :padding
  when "\x01"
    :ping
  when "\x02".."\x03"
    :ack
  # .....
```

Is it time for pattern matching? (This is a bad case. Too simple.)



But, I'm negative about introducing special class for bytes data

- It may break the existing code base
 - Ruby has 20+ years of history
 - "Bytes" is a very common noun, especially computer science
 - "bytes" gem was already taken 😞

So...

- Use another name of the String class
- Create own helper methods of bit operation
- Create extension library (if absolutely necessary)



Conclusion

- handling/manipulating binary data in Ruby is hard
 - than other languages that support bytes data as standard
 - bit operation is slow
- should convert data on encrypt/decrypt operations

Why did I choose the "Pure Ruby" way?

- To avoid problems coming from ractor or multithreading
 - Problems will appear near future...maybe... 😊

