JOSHUA LIEBOW-FEESER

# MOVE FAST AND DON'T BREAK THINGS

# ME

- Security @ Google
  - Work on open-source Rust!
- Secure Code Working Group

# THE THING

- Experimental networking stack (link thru transport)
- Pure Rust
- Low CPU usage
- Low binary footprint
- Platform-agnostic core + bindings

# “HIGH-PERFORMANCE”

# OUTLINE

- Design goals
- Background: Packet formats
- Part 1: Parsing
- Part 2: Serialization
- Part 3: Forwarding (Parsing + Serialization)
- Part 4: Zero-copy

# GOALS

- Zero copying
- Zero heap allocation
- Zero unsafe

# GOALS

- Zero copying
- Zero heap allocation
- Very little unsafe

# “PACKET”

# A NETWORK PACKET

ETHERNET PACKET

# A NETWORK PACKET

| ETHERNET HEADER | ETHERNET BODY |
| --- | --- |

# A NETWORK PACKET

| ETHERNET HEADER | IP PACKET |
| --- | --- |

# A NETWORK PACKET

| ETHERNET HEADER | IP HEADER | IP BODY |
|---|---|---|

# A NETWORK PACKET

| ETHERNET HEADER | IP HEADER | UDP PACKET |
|---|---|---|

# END GOAL (PART 3)

- Allocate a buffer on the stack, receive an Ethernet packet
- Parse the Ethernet packet
- Parse the IP packet
- Decide to forward the packet
- Update the IP header
- Serialize the Ethernet header

PART 1

# PARSING

# GOAL

- Allocate a buffer on the stack, receive an Ethernet packet
- Parse the Ethernet packet
- Parse the IP packet
- Process the UDP packet

# THE BUFFER TRAIT

BUFFER

# THE BUFFER TRAIT

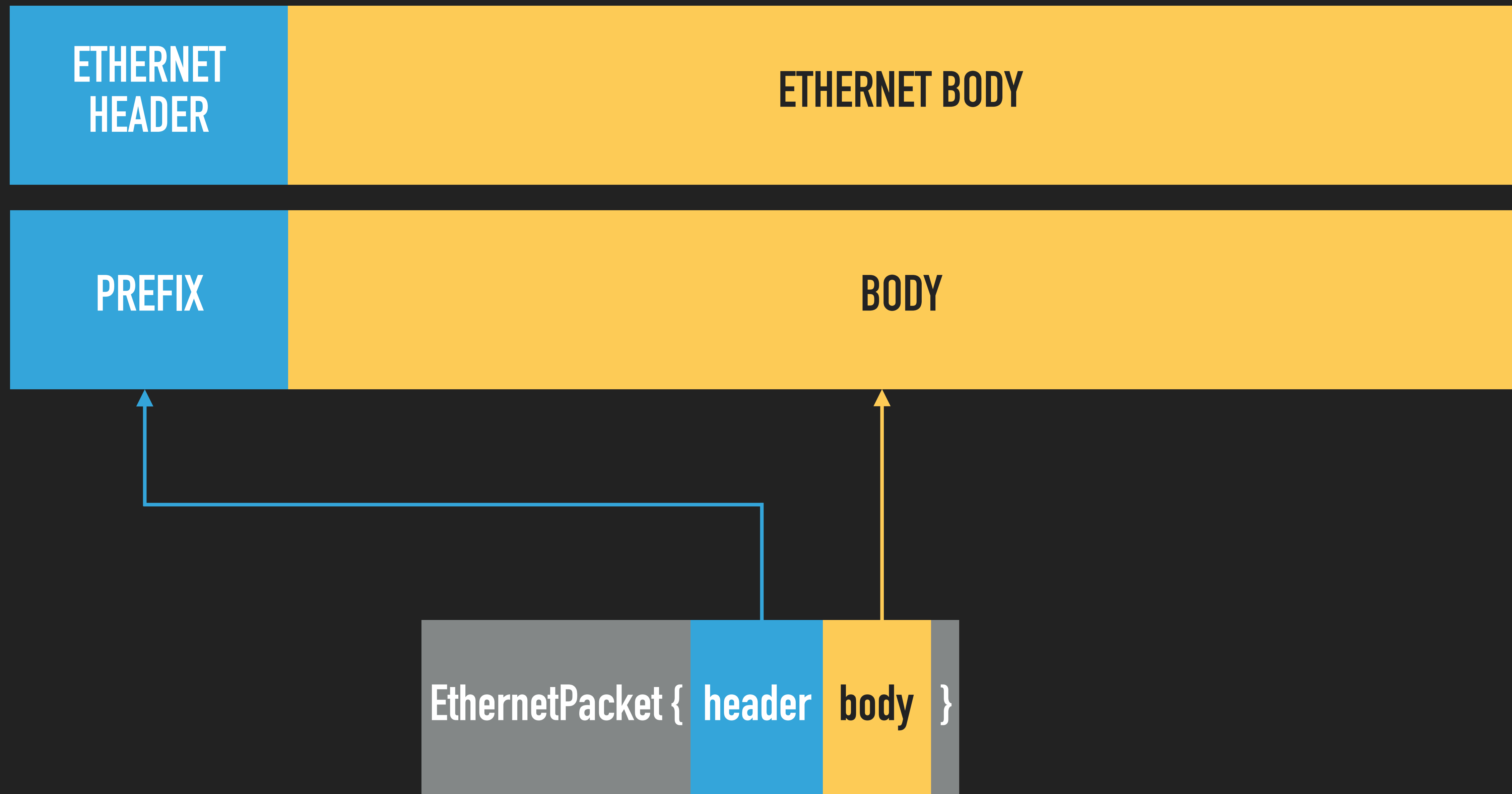
| PREFIX | BODY |
| --- | --- |

# PARSING FROM A BUFFER

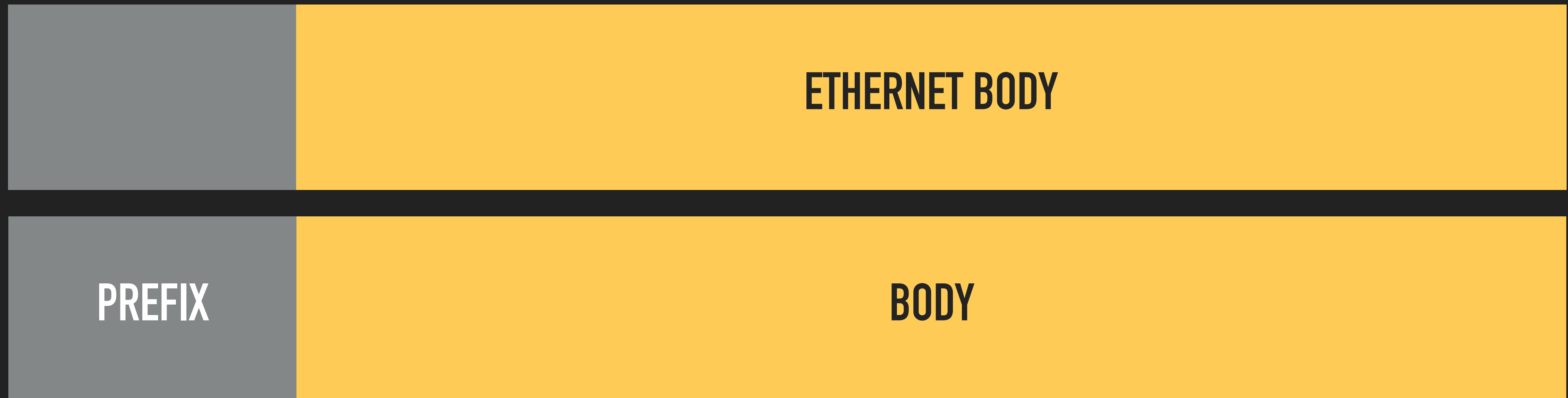
| ETHERNET HEADER | ETHERNET BODY |
|---|---|

| BODY |
|---|

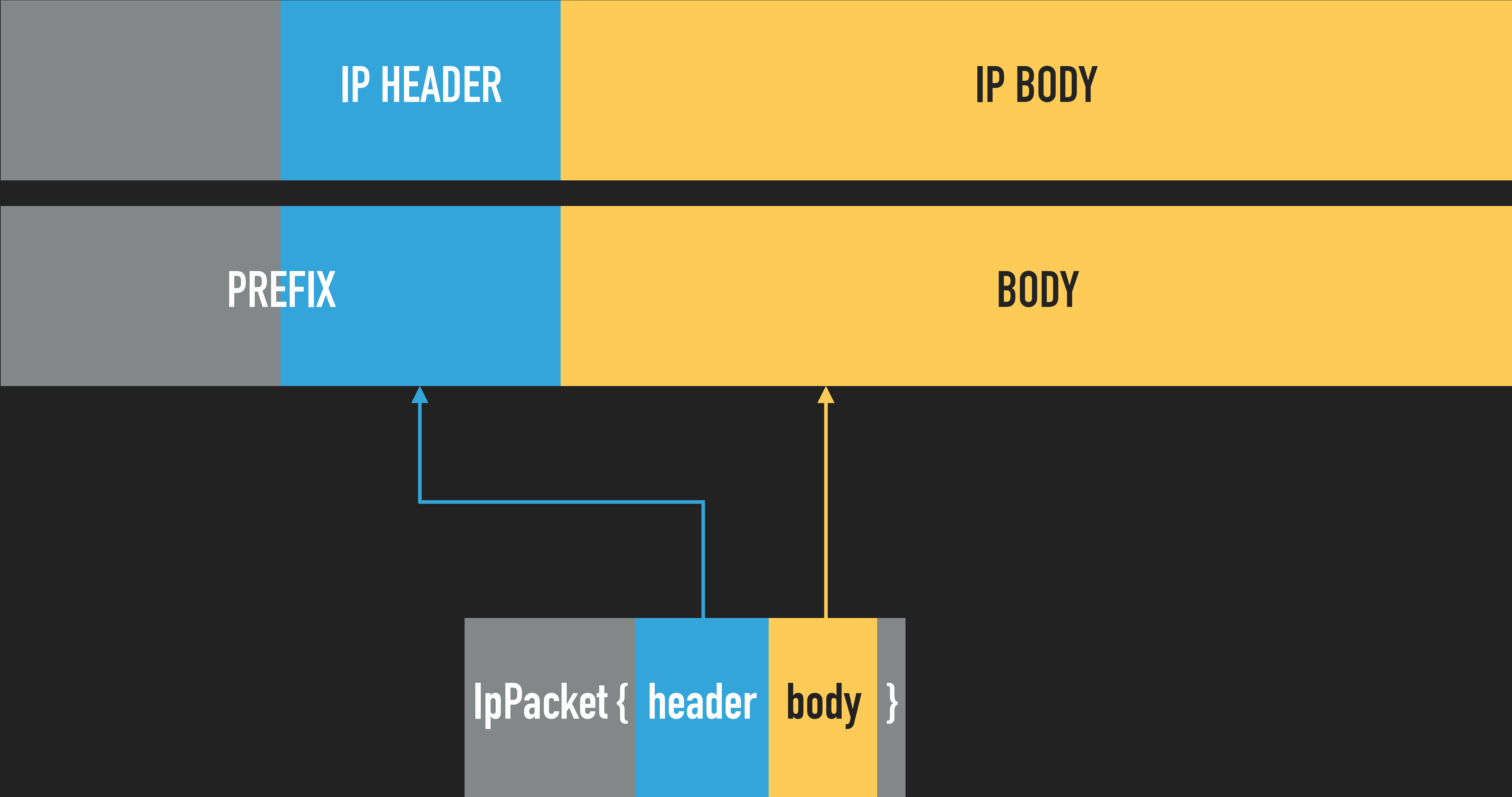
# PARSING FROM A BUFFER



# PARSING FROM A BUFFER



# PARSING FROM A BUFFER



# PARSING FROM A BUFFER

```
fn receive_ethernet_packet<B: Buffer>(mut buffer: B) {
    let packet = buffer.parse::<EthernetPacket>()?;
    ip::receive_ip_packet(buffer);
}
```

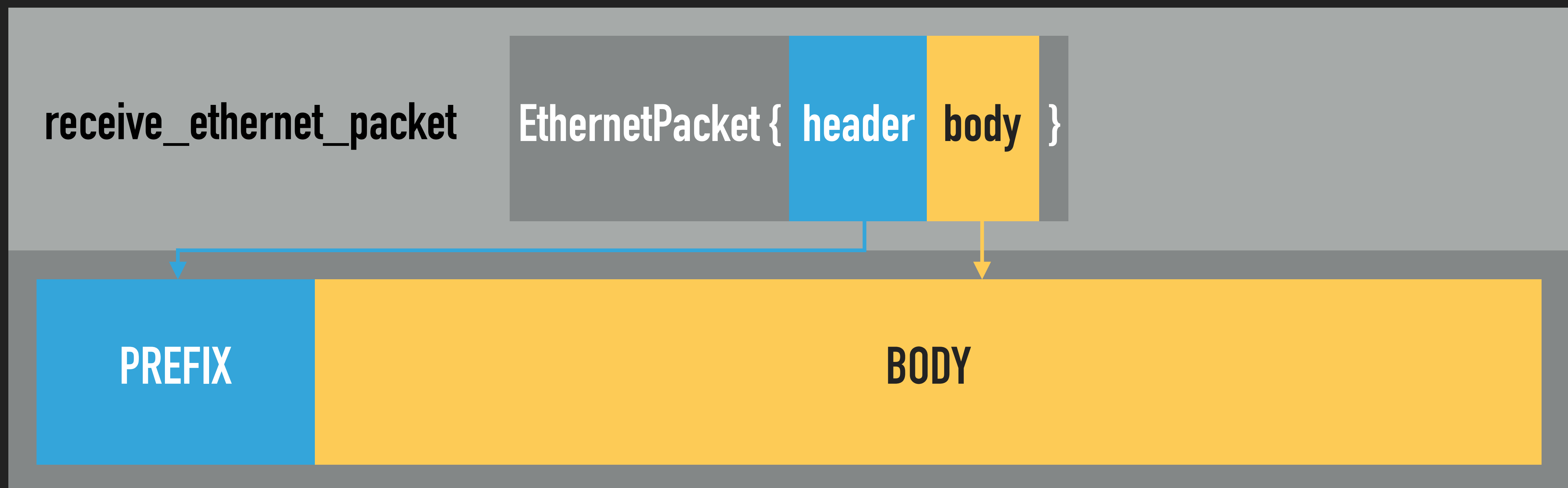
# PARSING FROM A BUFFER

# PARSING FROM A BUFFER

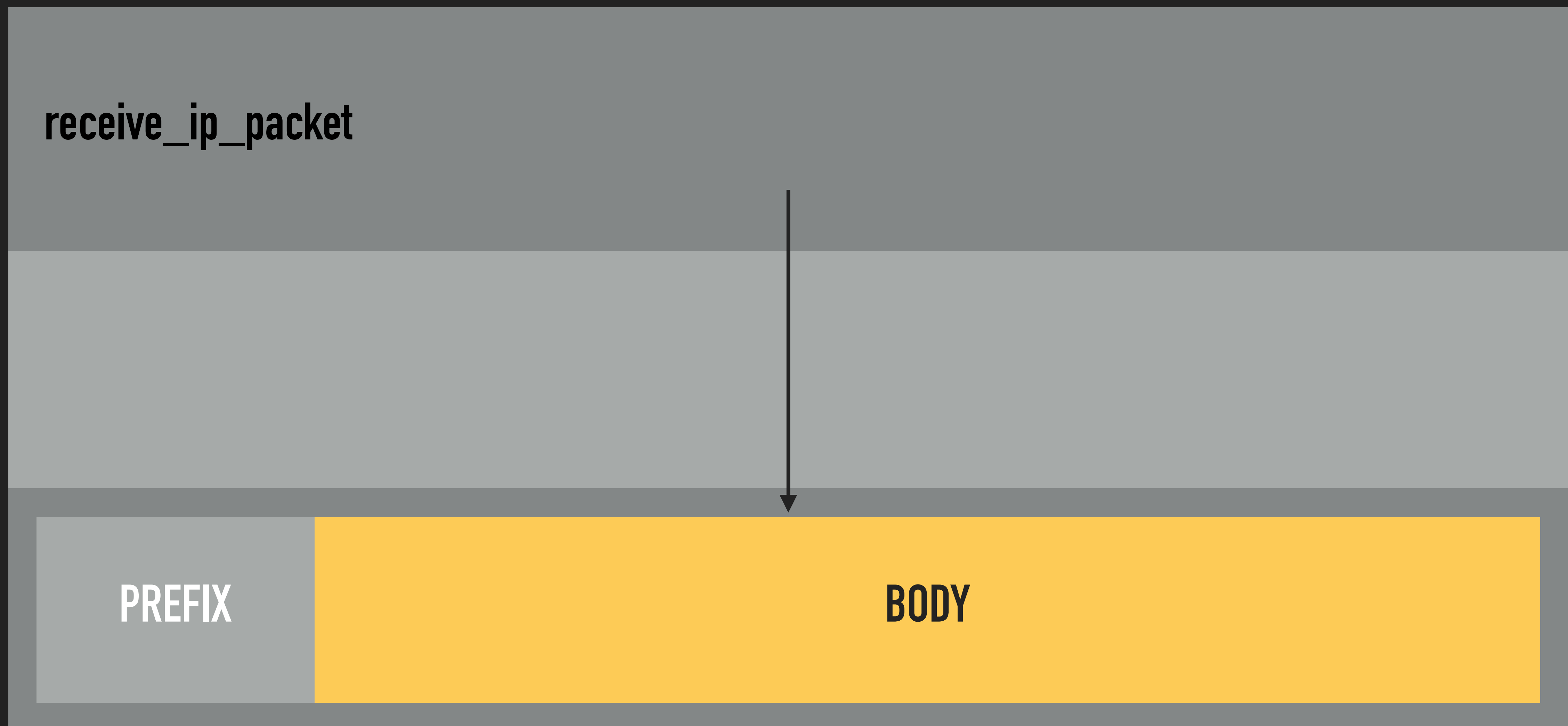
receive_ethernet_packet

BODY

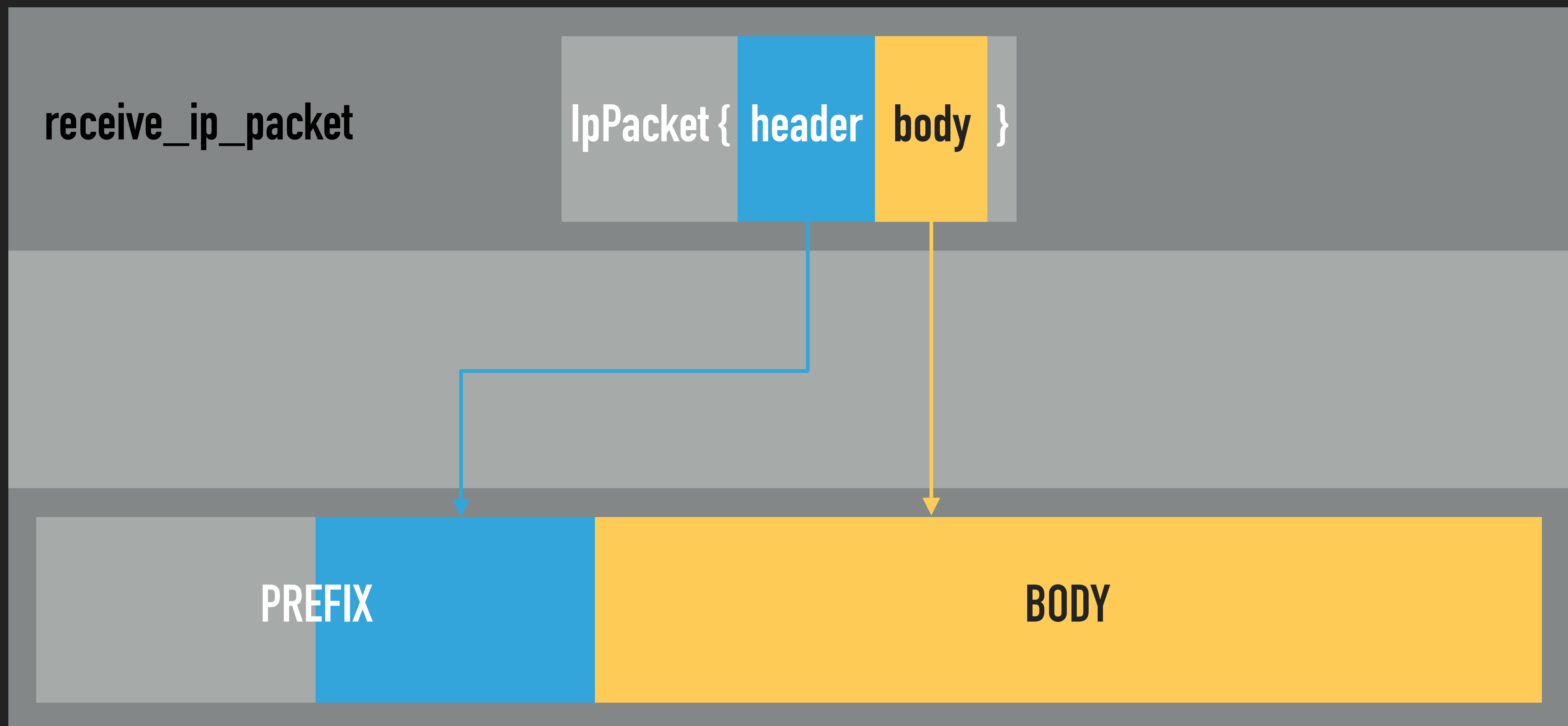
# PARSING FROM A BUFFER



# PARSING FROM A BUFFER



# PARSING FROM A BUFFER



PART 2

# SERIALIZATION

# GOAL

- Receive request to send a UDP packet
- Compute UDP header information
- Compute IP routing and header information
- Compute Ethernet routing and header information
- Compute length, and allocate a buffer
- Serialize UDP body
- Serialize UDP header
- Serialize IP header
- Serialize Ethernet header

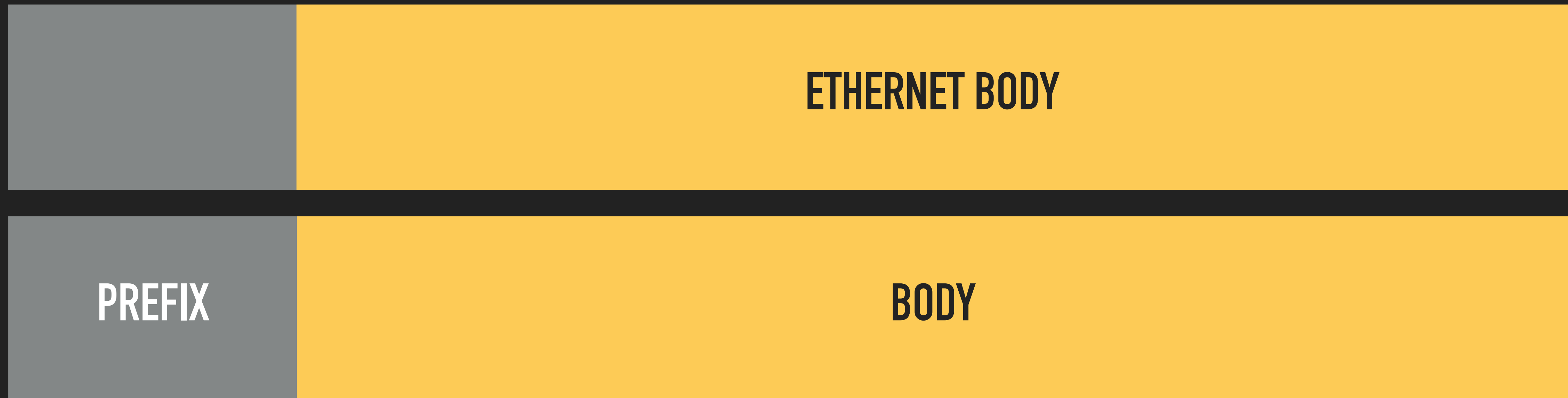
## WHY IS THIS HARD?

- Need to send request to next layer *before* serializing

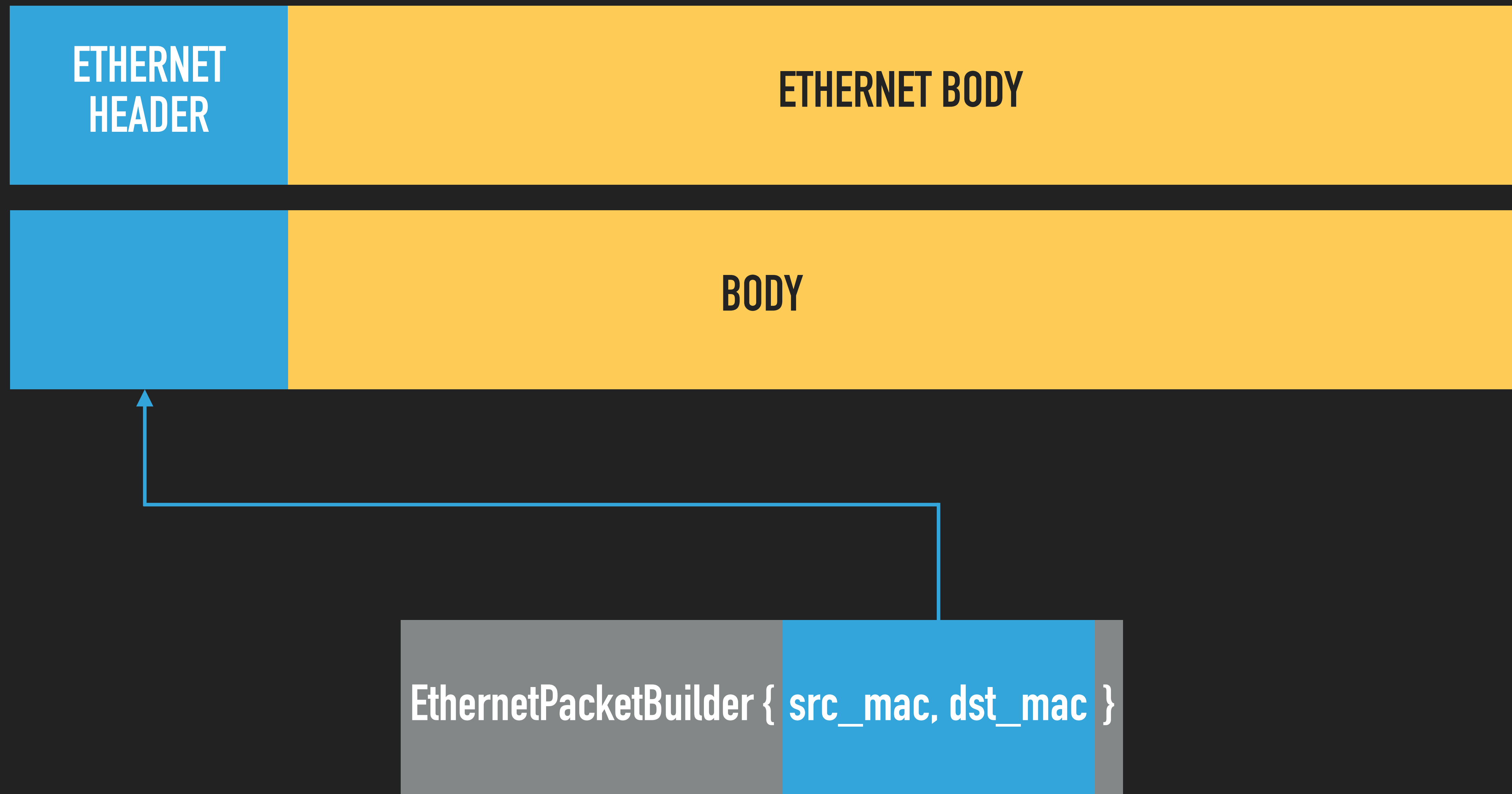
# THE PACKETBUILDER TRAIT

```
trait PacketBuilder {
    fn header_len(&self) -> usize;
    fn serialize<B: Buffer>(self, buffer: &mut B);
}
```

# SERIALIZING INTO A BUFFER



# SERIALIZING INTO A BUFFER



# THE SERIALIZER TRAIT

```
trait Serializer {
    type Buffer: Buffer;
}
```

# THE SERIALIZER TRAIT

```
fn encapsulate_ip_ethernet<S: Serializer>(ser: S) ->
    impl Serializer {
    let ip = ser.encapsulate(IpPacketBuilder::new(...));
    ip.encapsulate(EthernetPacketBuilder::new(...))
}
```

# THE SERIALIZER TRAIT

```
fn encapsulate_ip_ethernet<S: Serializer>(ser: S) ->
     impl Serializer {
     let ip = ser.encapsulate(IpPacketBuilder::new(...));
     ip.encapsulate(EthernetPacketBuilder::new(...))
}
```

# THE SERIALIZER TRAIT

```
fn encapsulate_ip_ethernet<S: Serializer>(ser: S) ->
    impl Serializer {
    let ip = ser.encapsulate(IpPacketBuilder::new(...));
    ip.encapsulate(EthernetPacketBuilder::new(...))
}
```

# THE SERIALIZER TRAIT

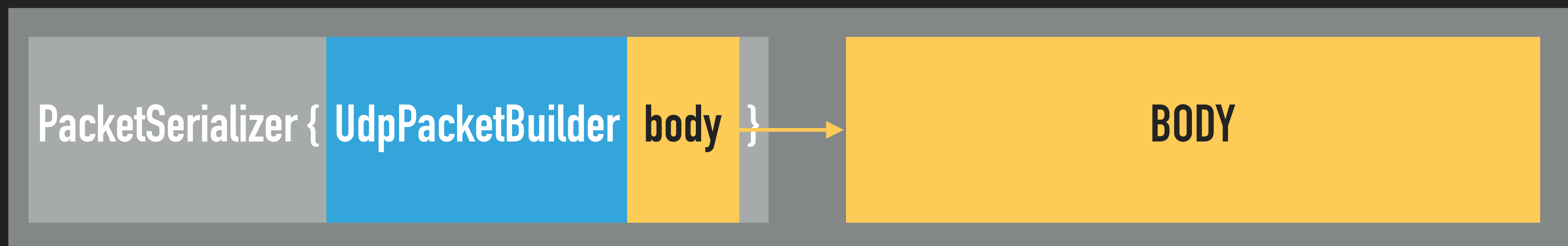
```
fn encapsulate_ip_ethernet<S: Serializer>(ser: S) ->
    impl Serializer {
    let ip = ser.encapsulate(IpPacketBuilder::new(...));
    ip.encapsulate(EthernetPacketBuilder::new(...))
}
```

EncapsulatingSerializer { EncapsulatingSerializer { ser IpPacketBuilder } EthernetPacketBuilder }

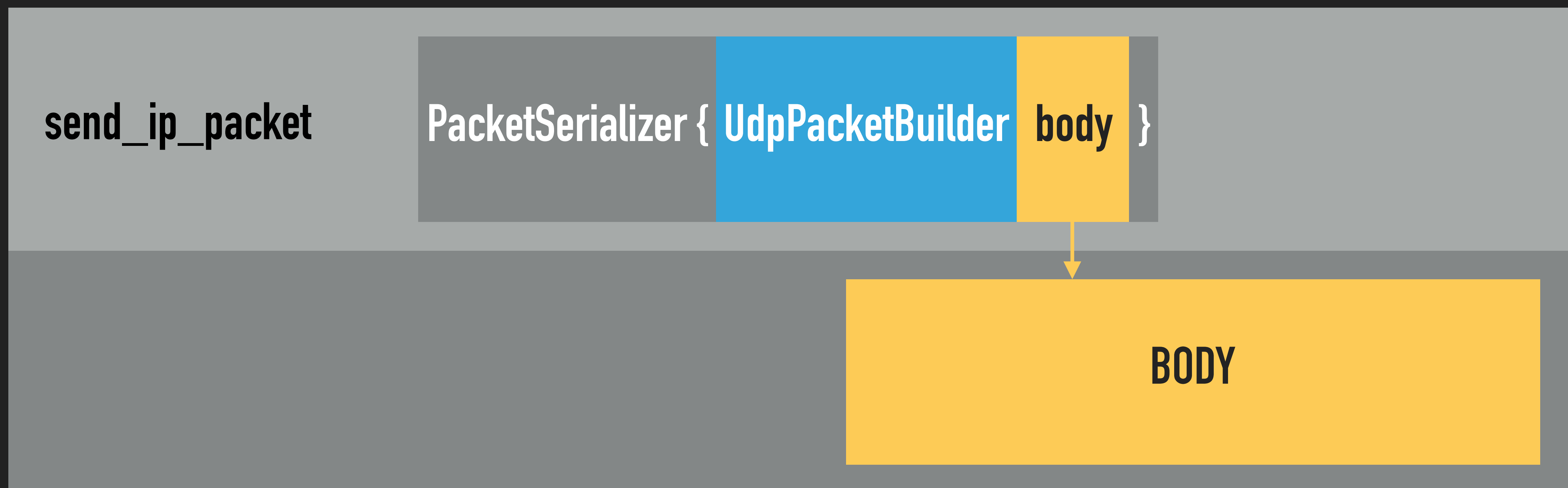
# THE SERIALIZER TRAIT

```
trait Serializer {
    type Buffer: Buffer;
    fn serialize(self, prefix_len: usize) -> Self::Buffer;
}
```

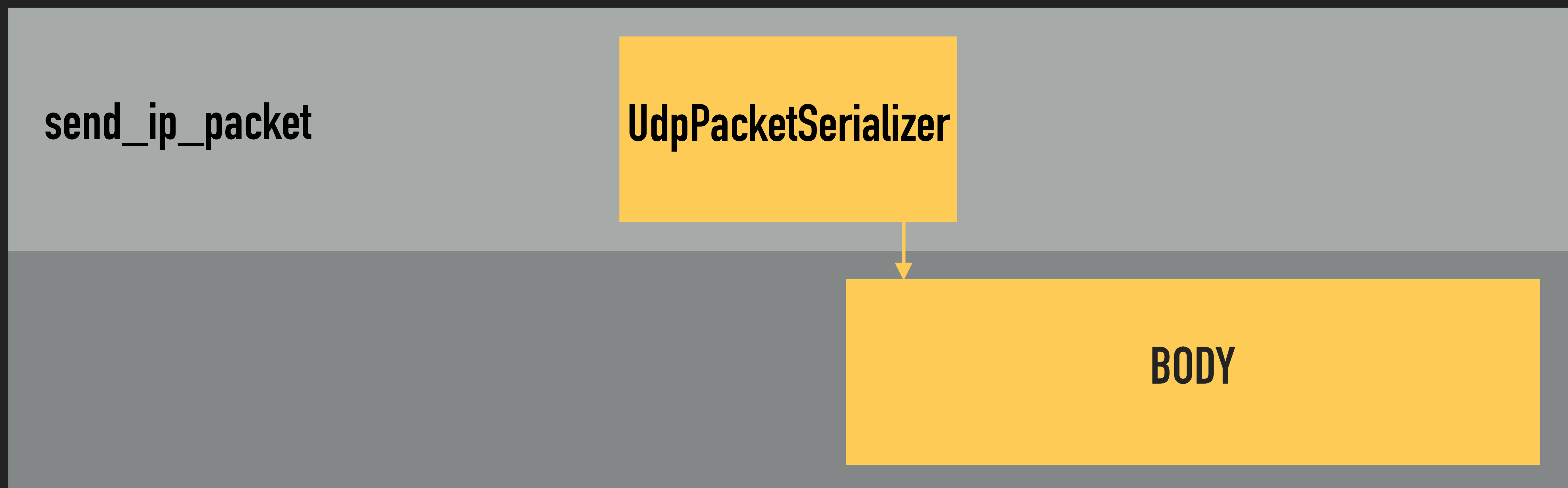
# THE SERIALIZER TRAIT



# THE SERIALIZER TRAIT

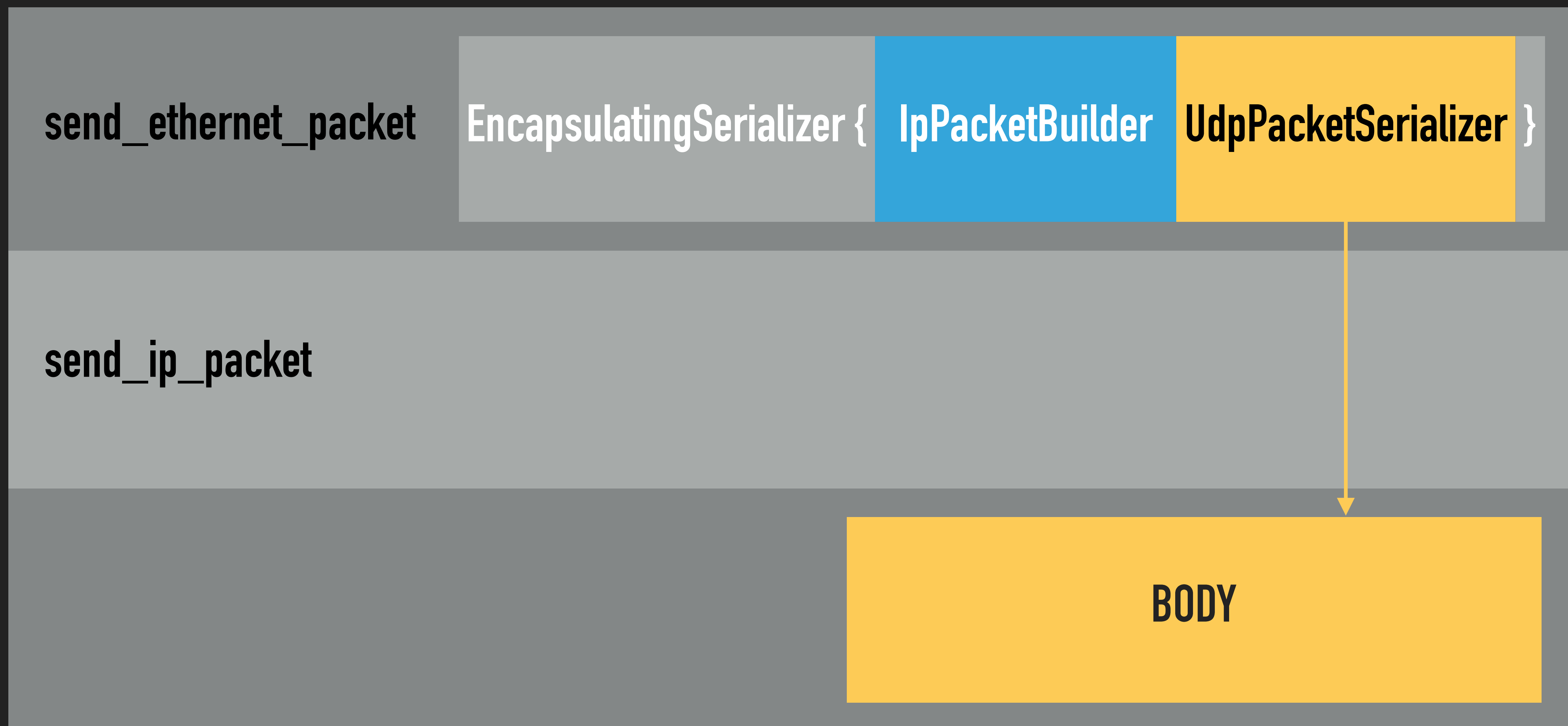


# THE SERIALIZER TRAIT

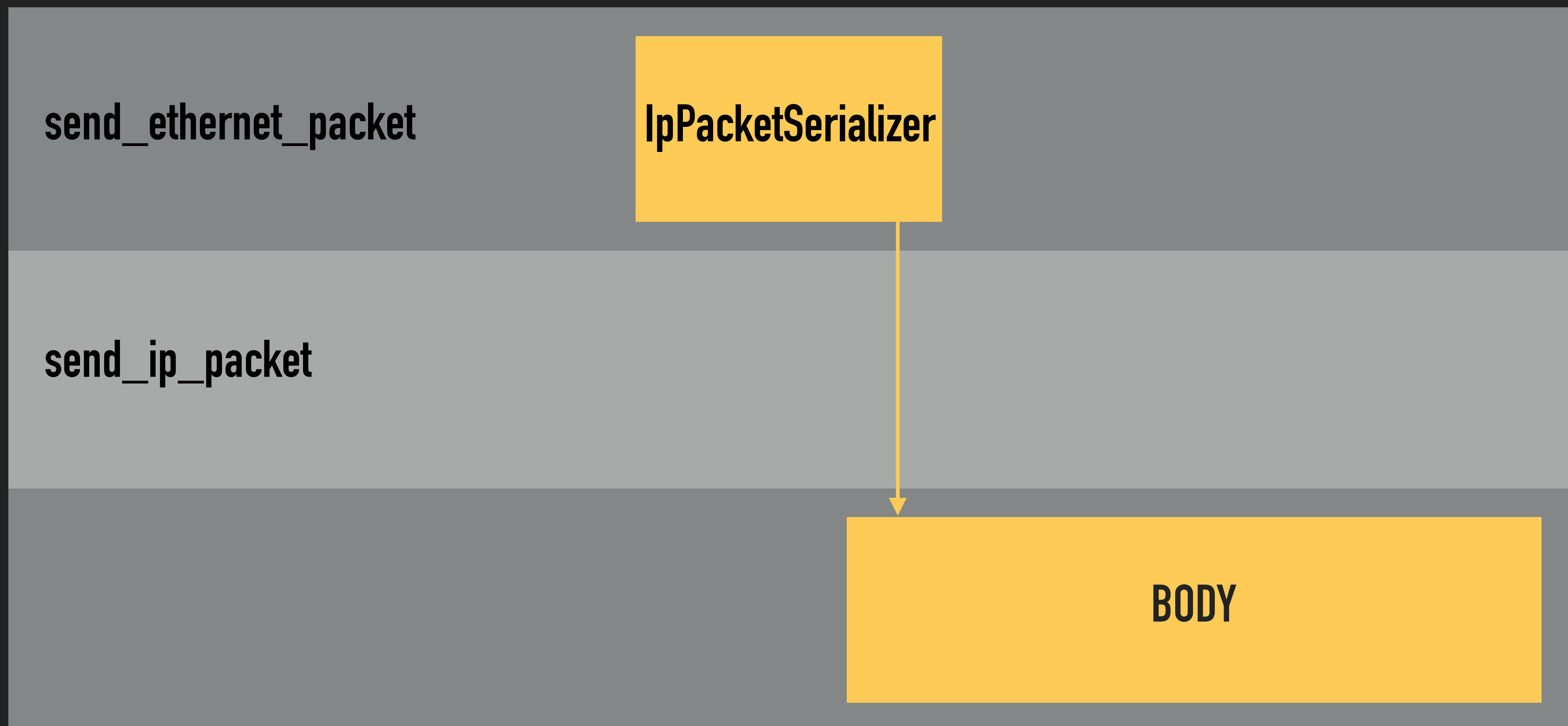


# THE SERIALIZER TRAIT

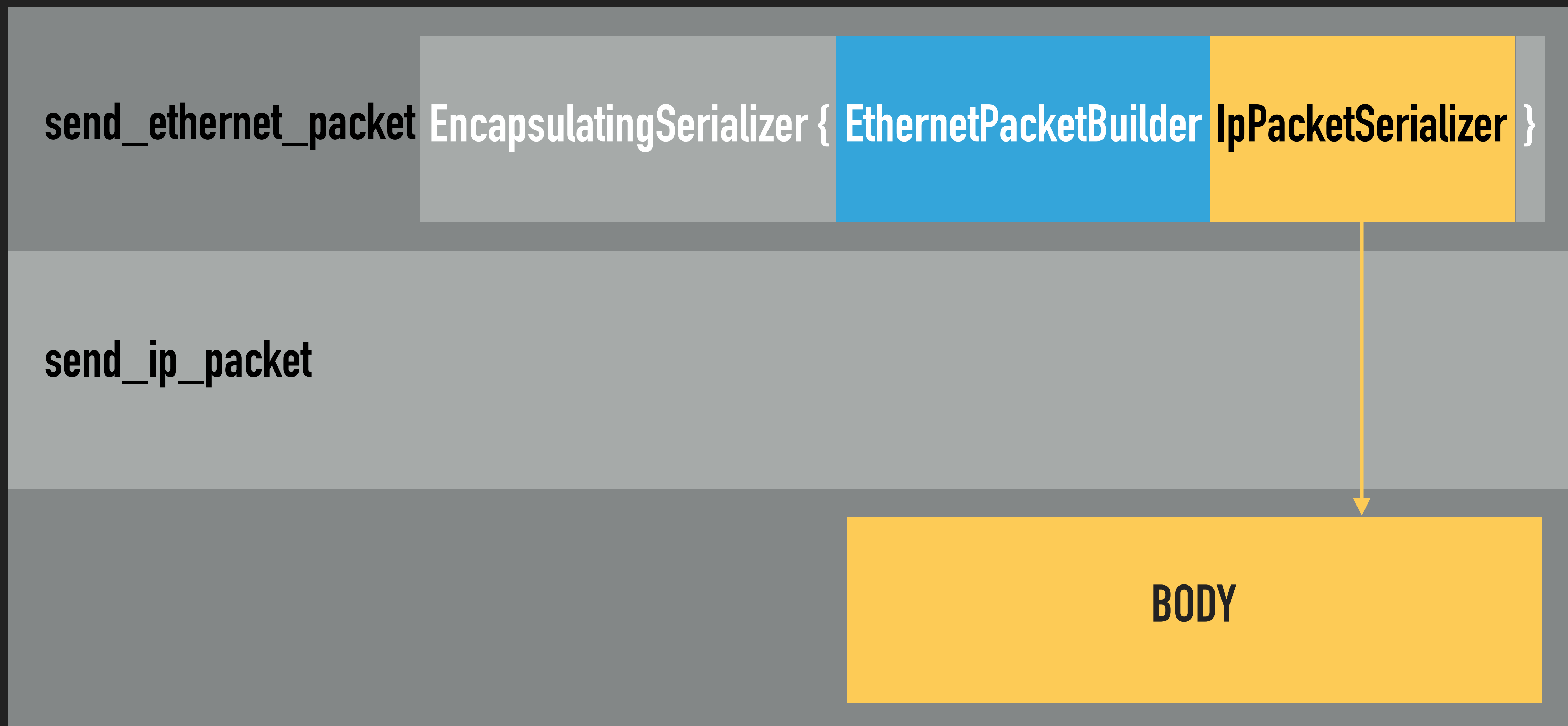
# THE SERIALIZER TRAIT



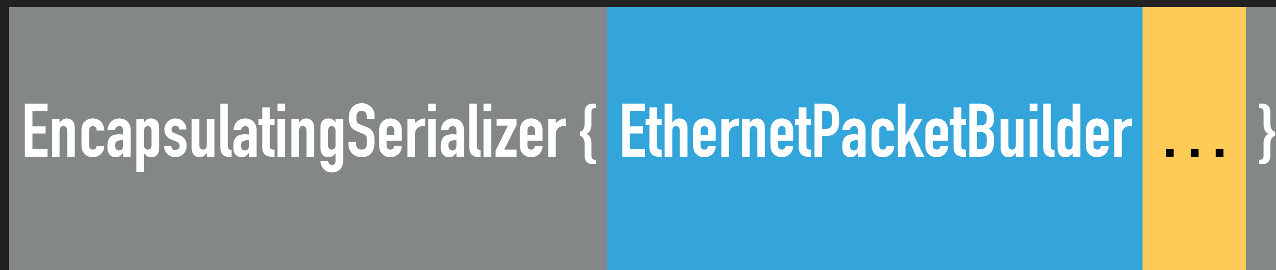
# THE SERIALIZER TRAIT



# THE SERIALIZER TRAIT



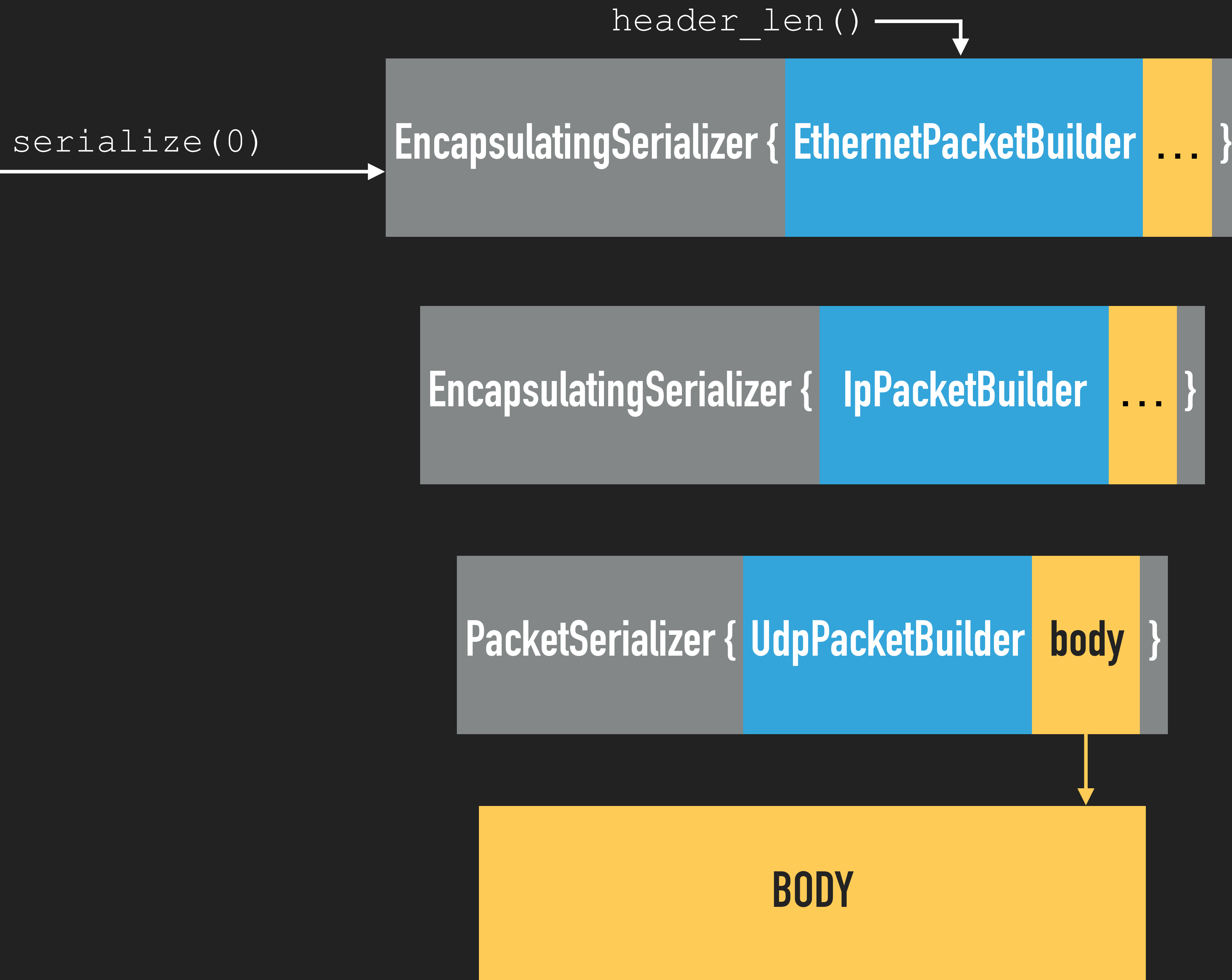
# THE SERIALIZER TRAIT



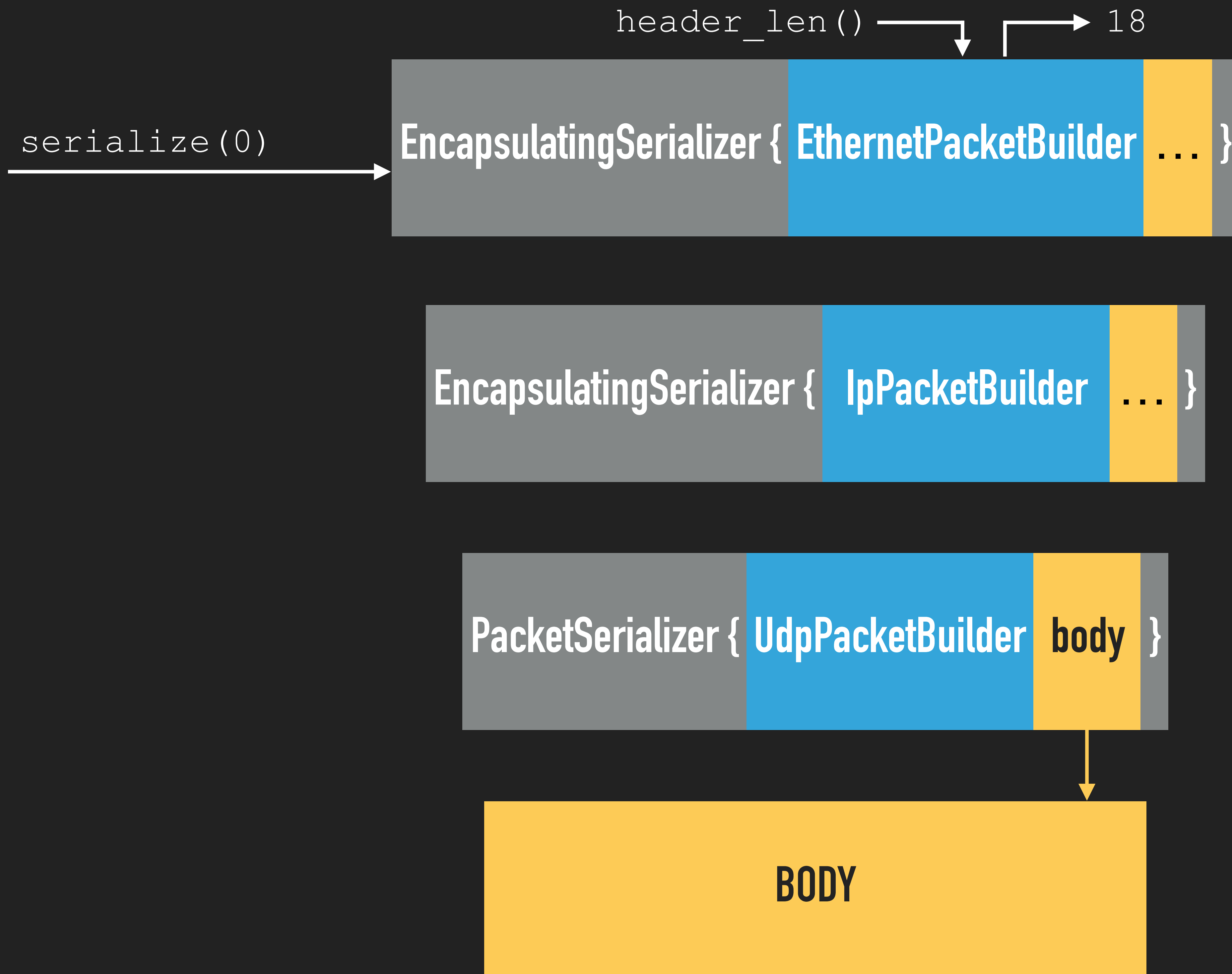
# THE SERIALIZER TRAIT

`serialize(0)`

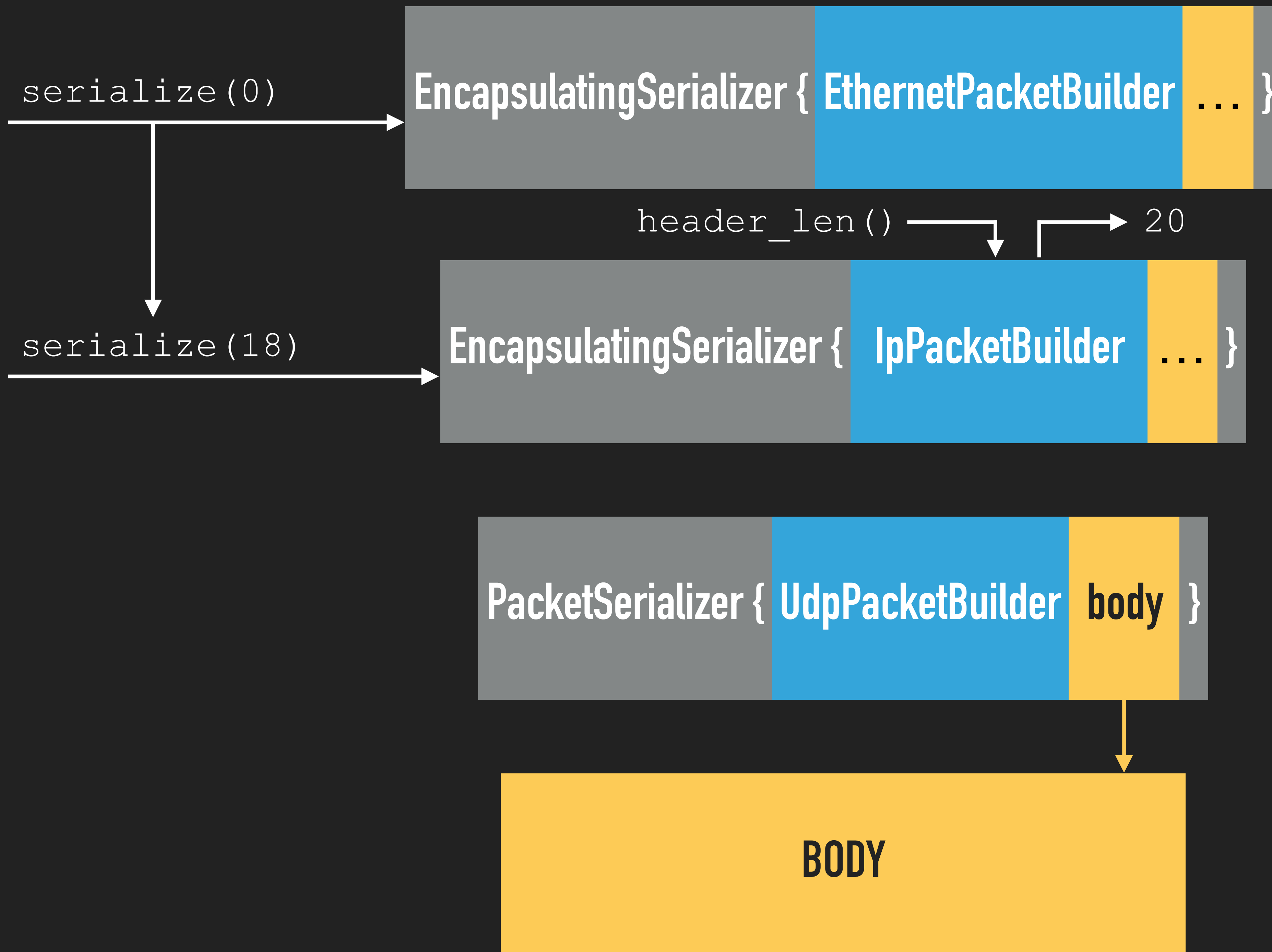
# THE SERIALIZER TRAIT



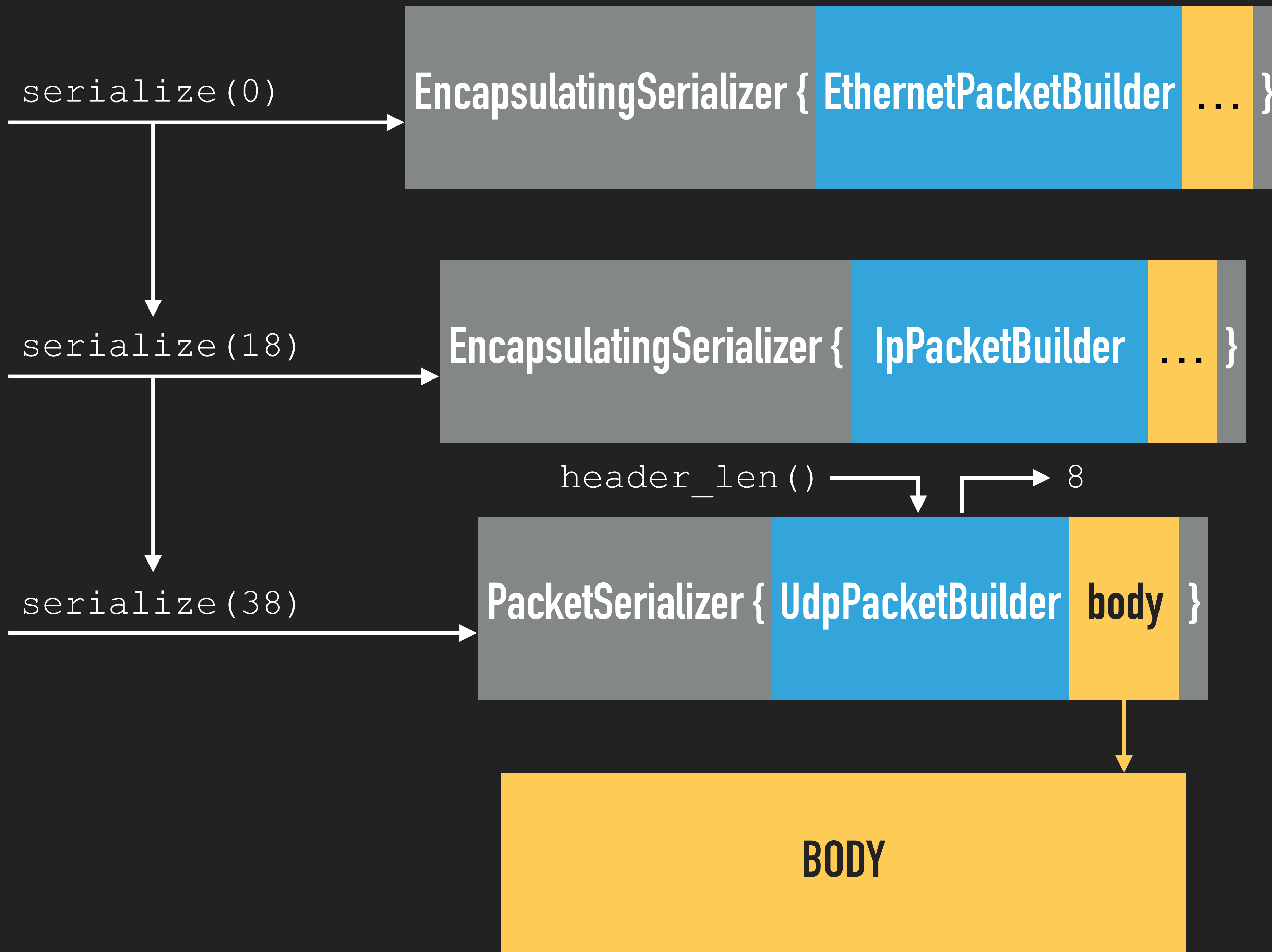
# THE SERIALIZER TRAIT



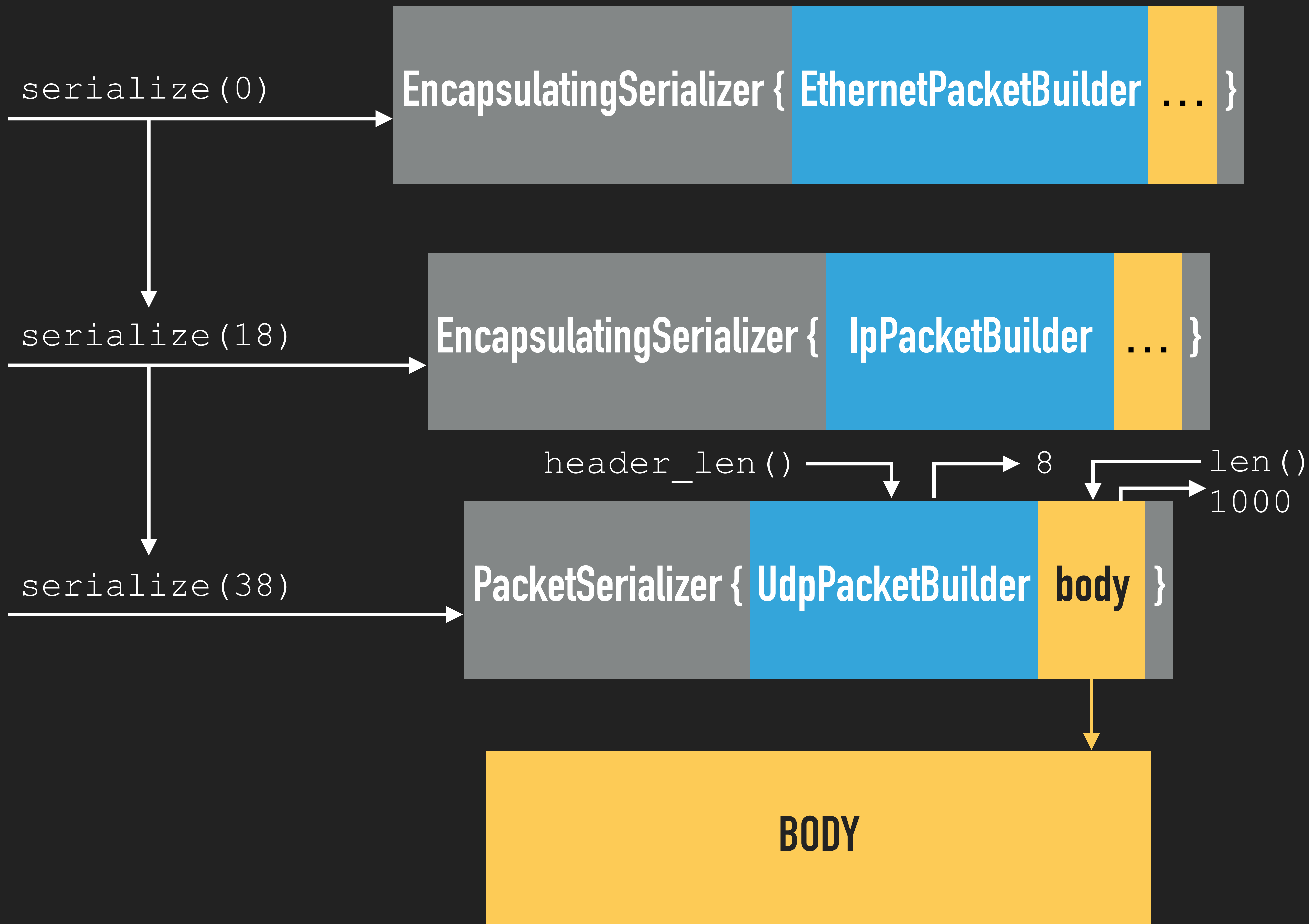
# THE SERIALIZER TRAIT



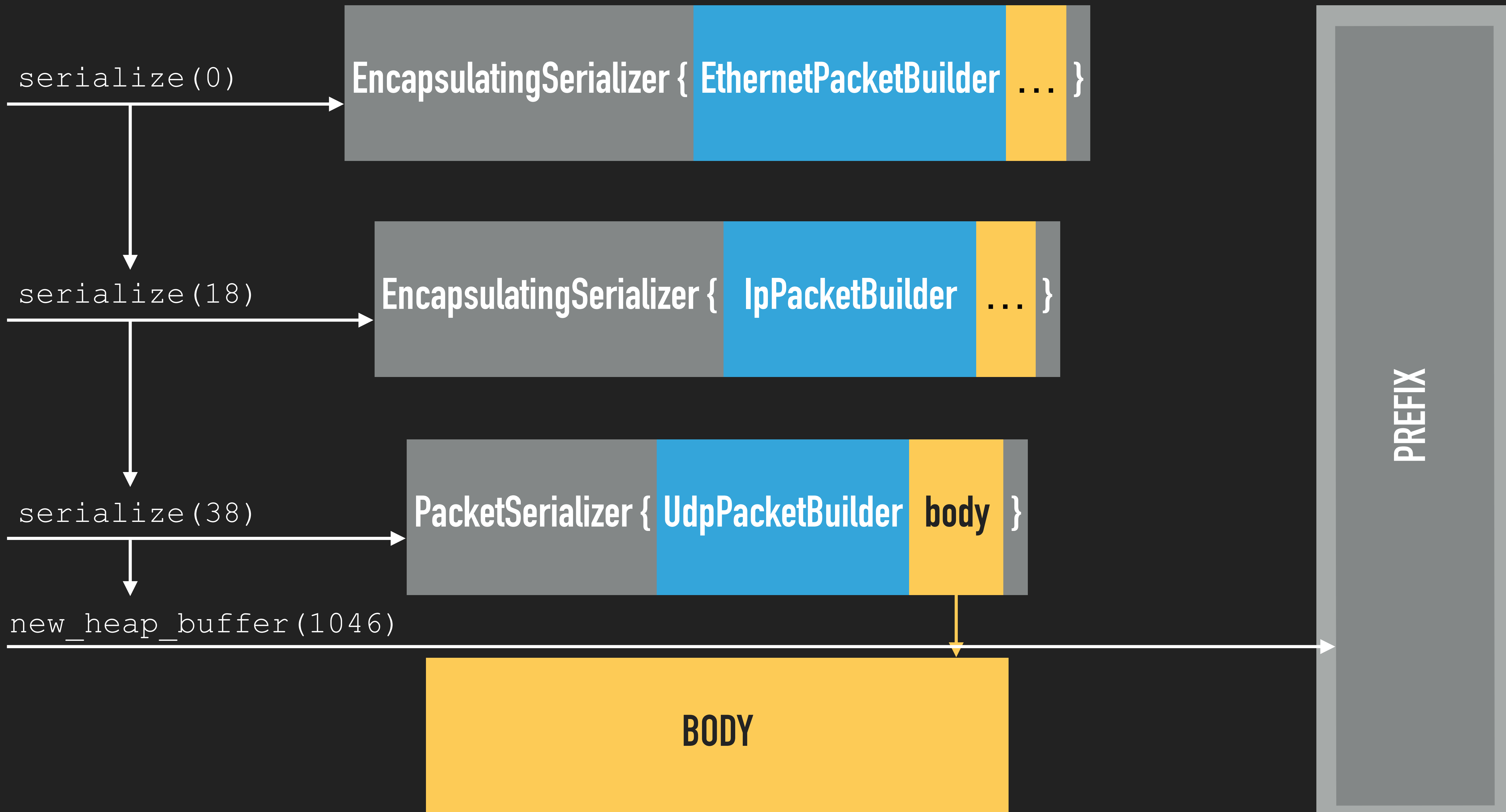
# THE SERIALIZER TRAIT



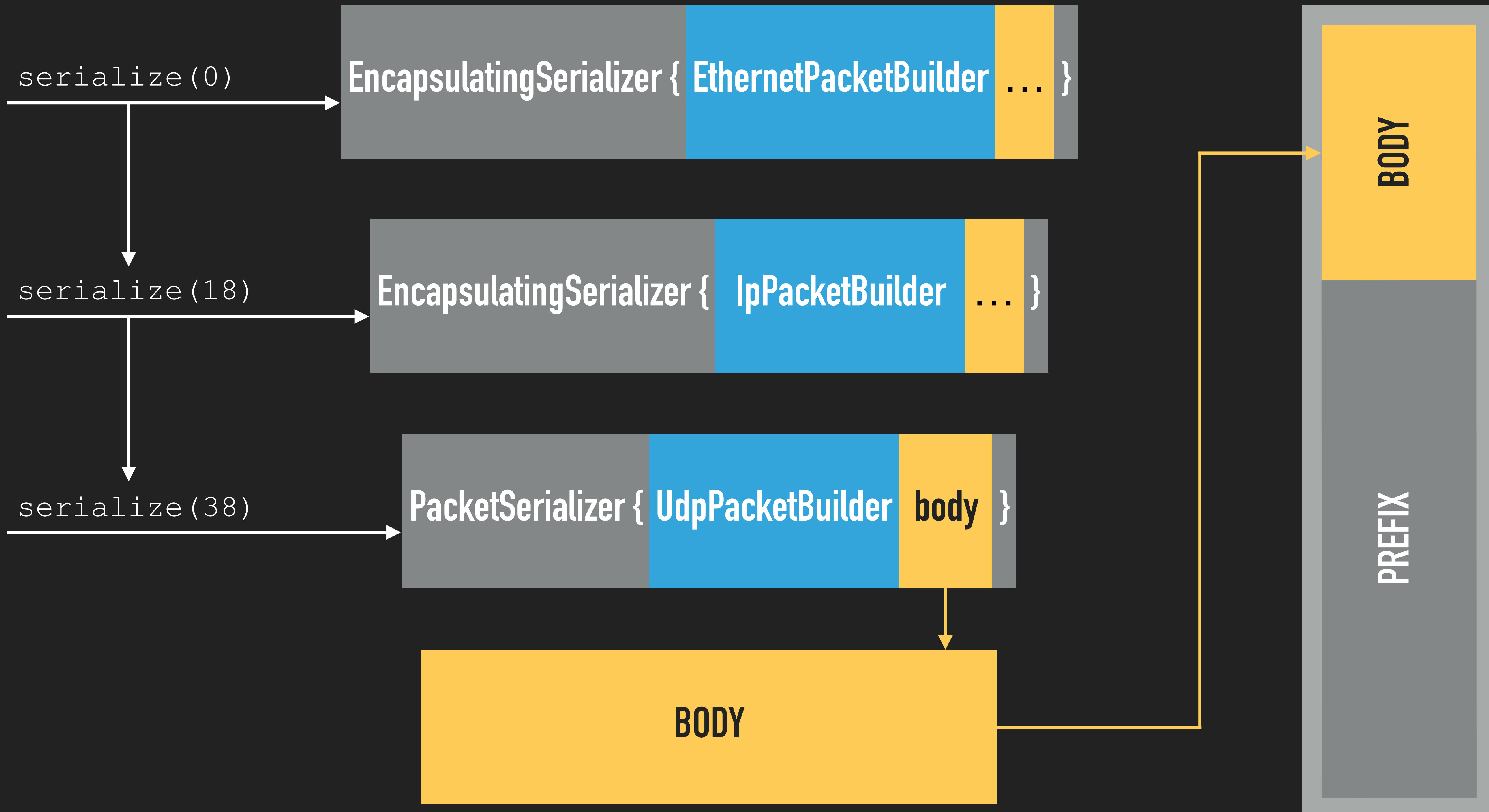
# THE SERIALIZER TRAIT



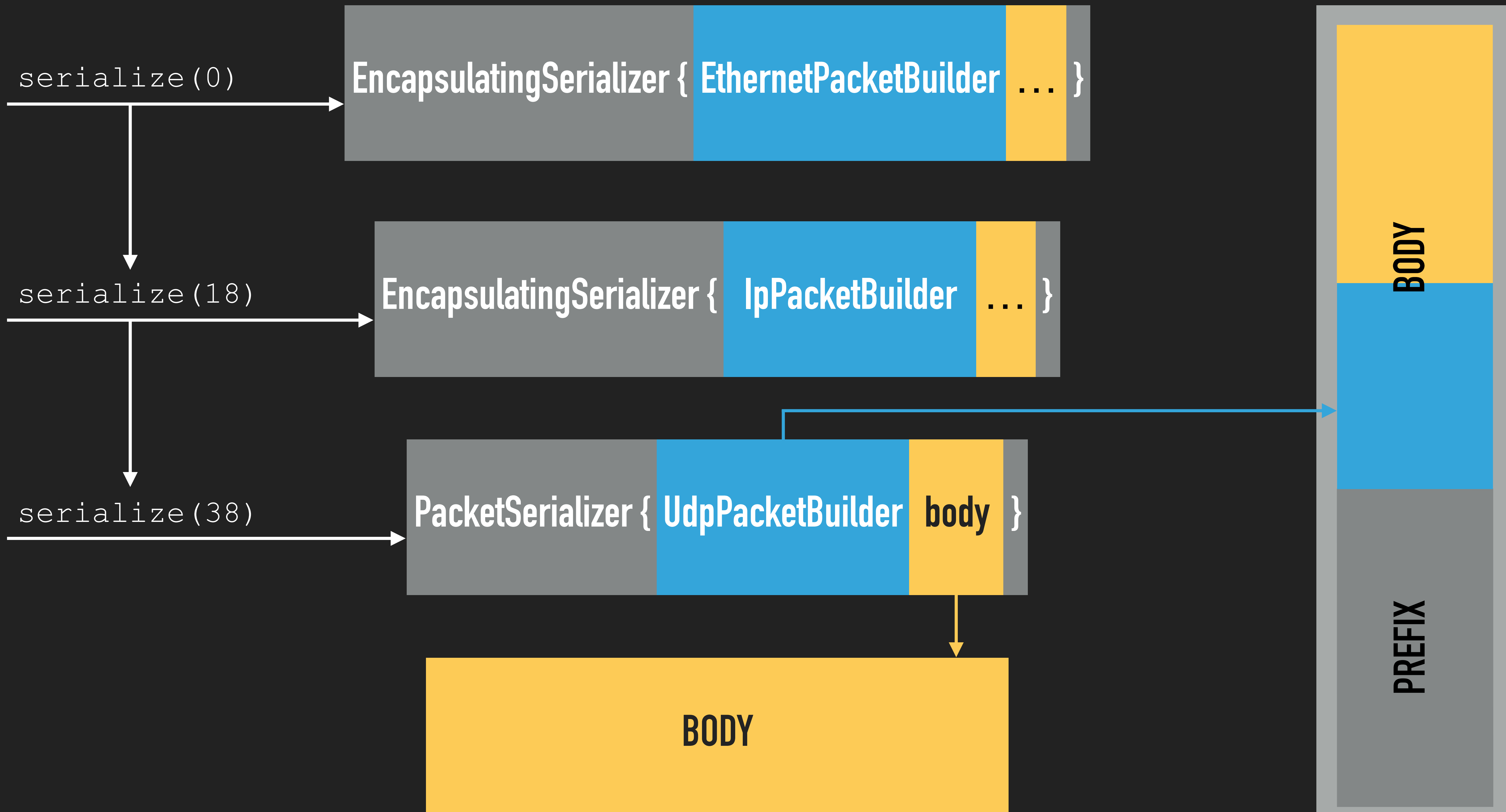
# THE SERIALIZER TRAIT



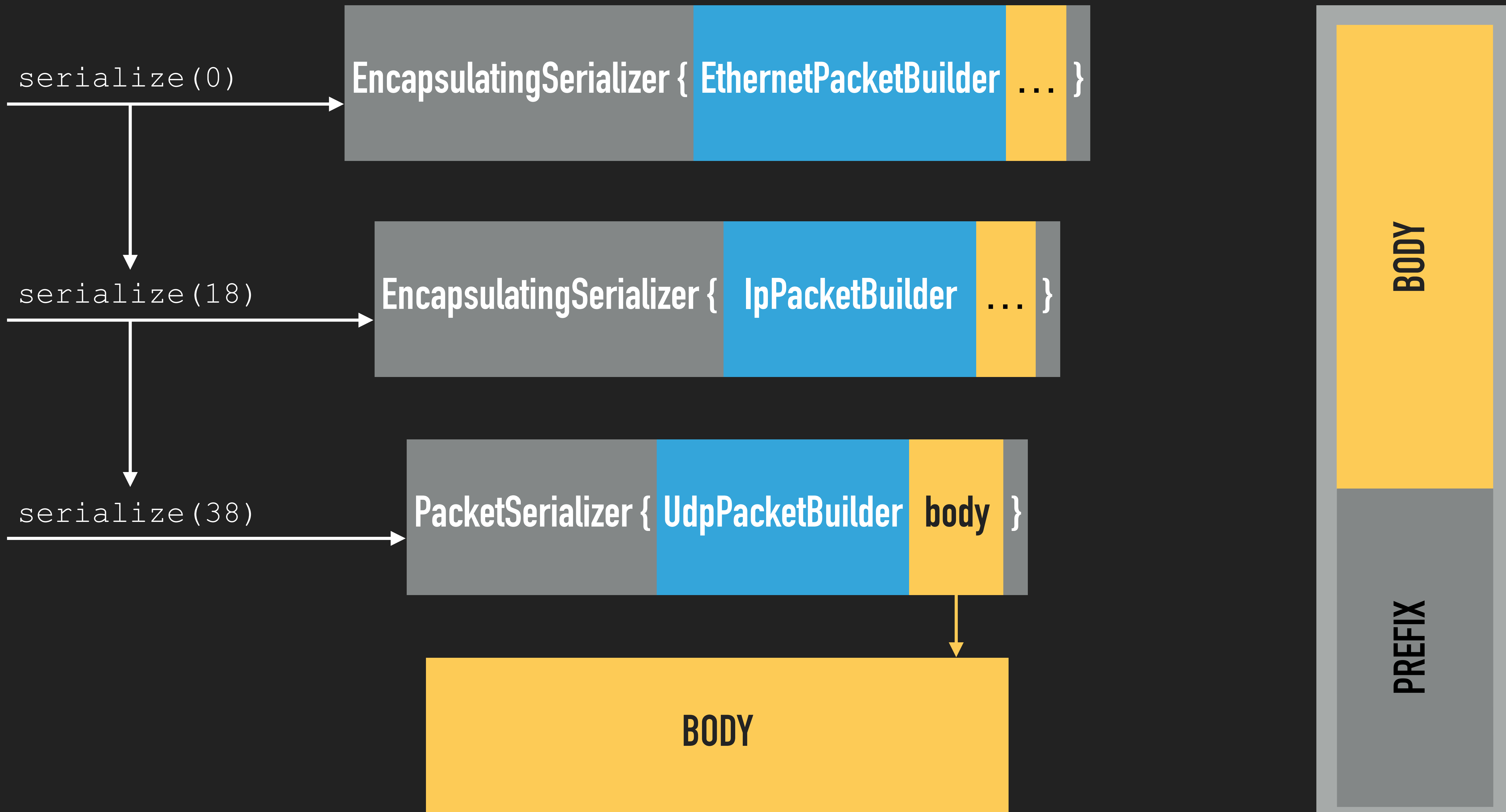
# THE SERIALIZER TRAIT



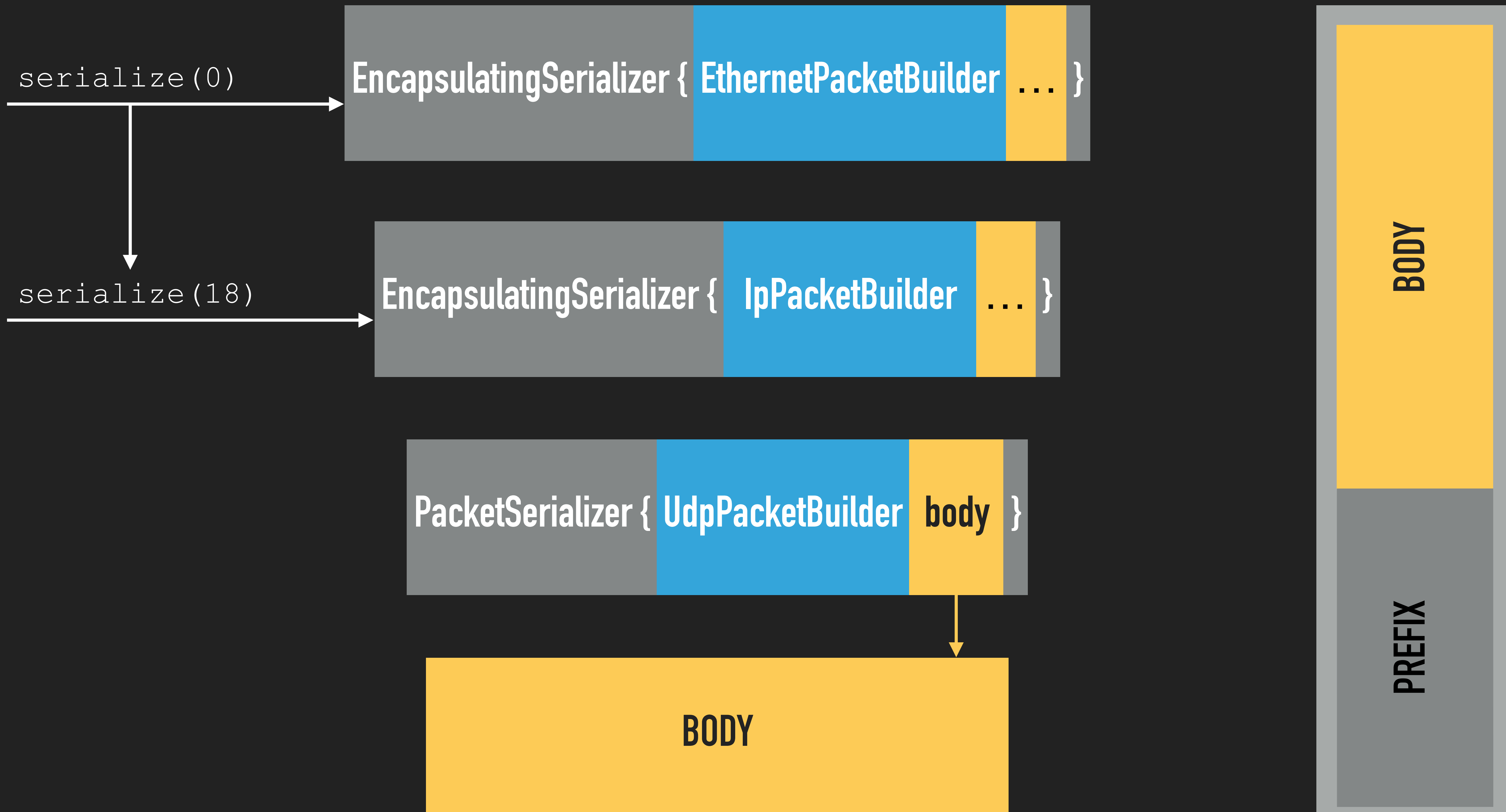
# THE SERIALIZER TRAIT



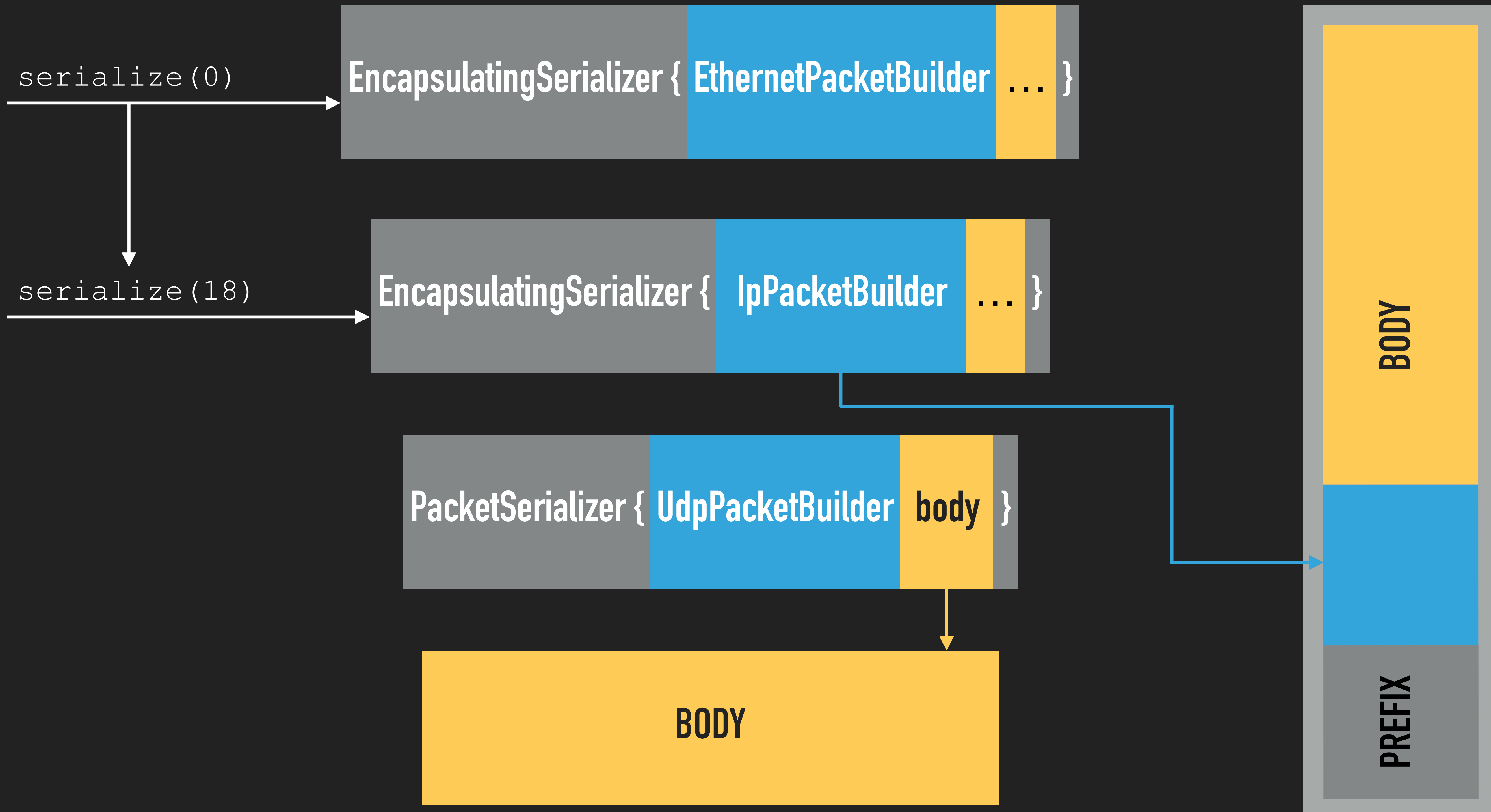
# THE SERIALIZER TRAIT



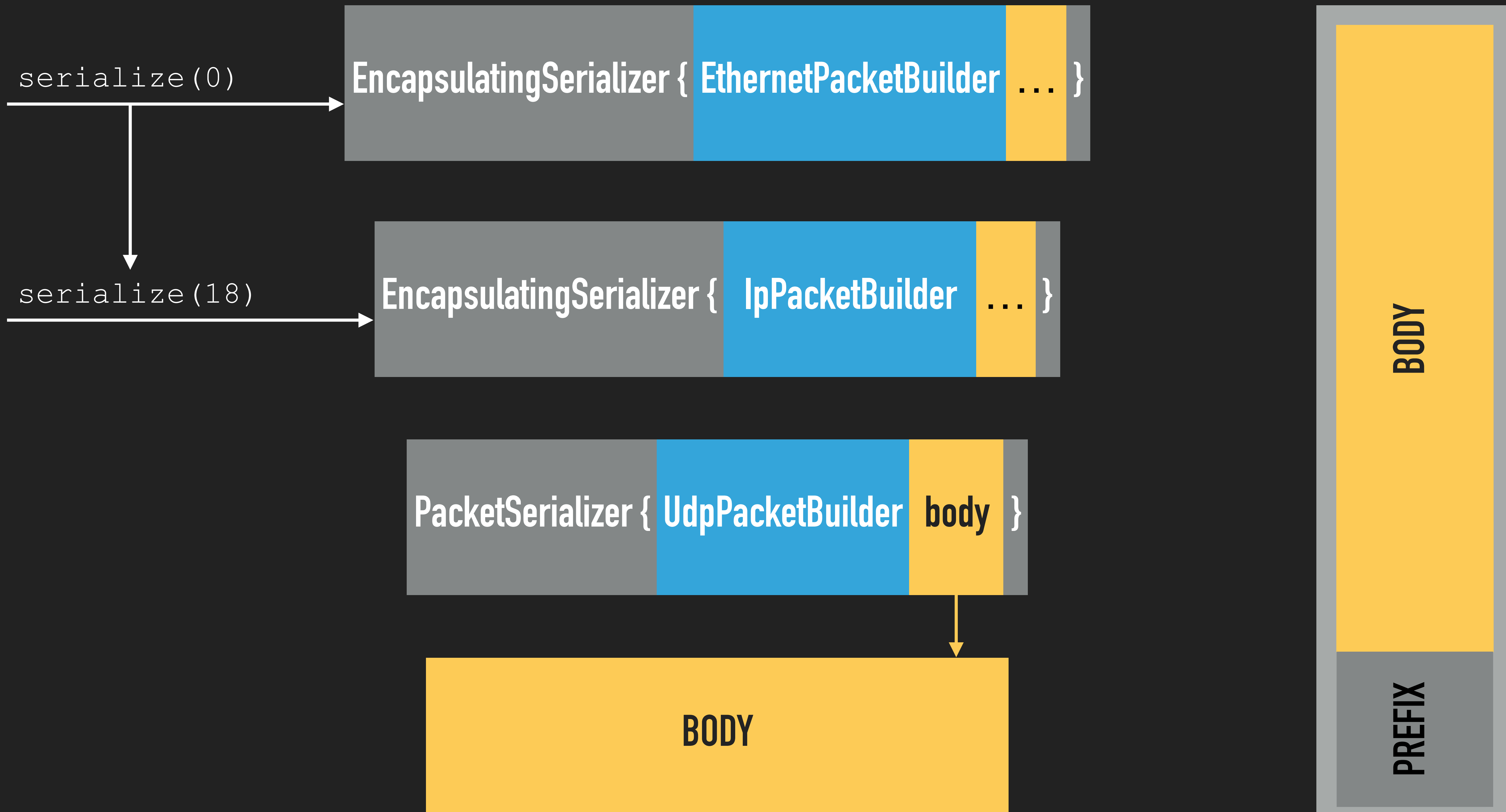
# THE SERIALIZER TRAIT



# THE SERIALIZER TRAIT

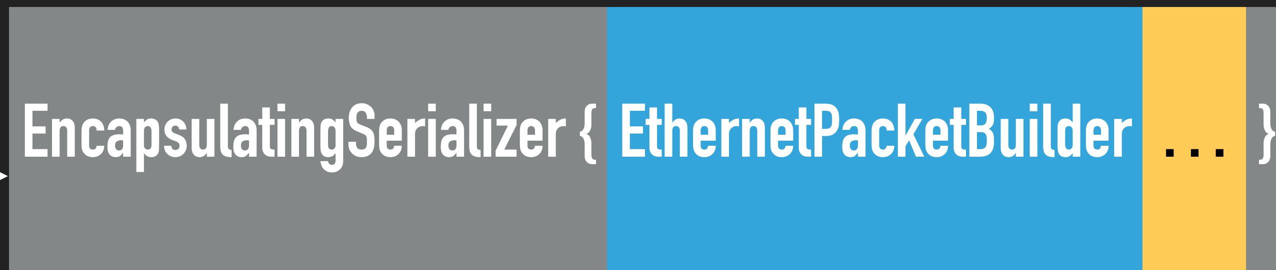


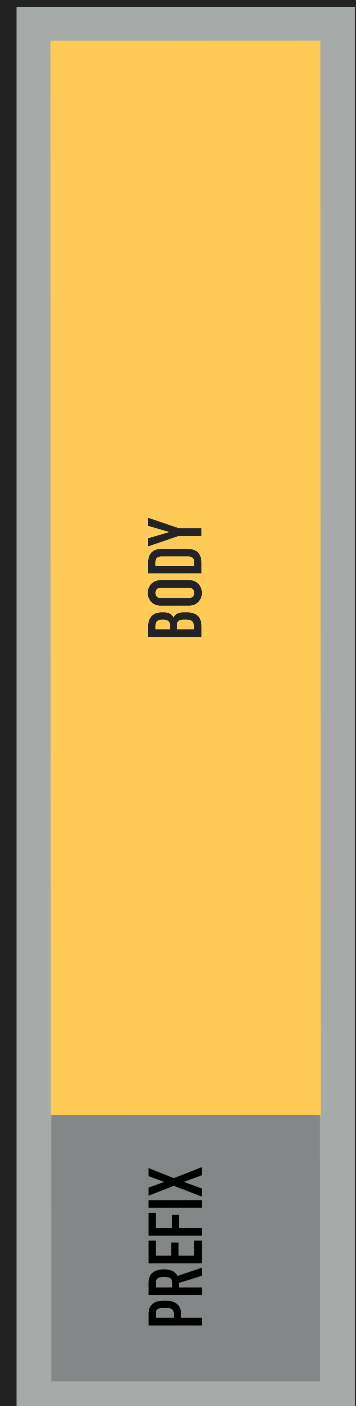
# THE SERIALIZER TRAIT



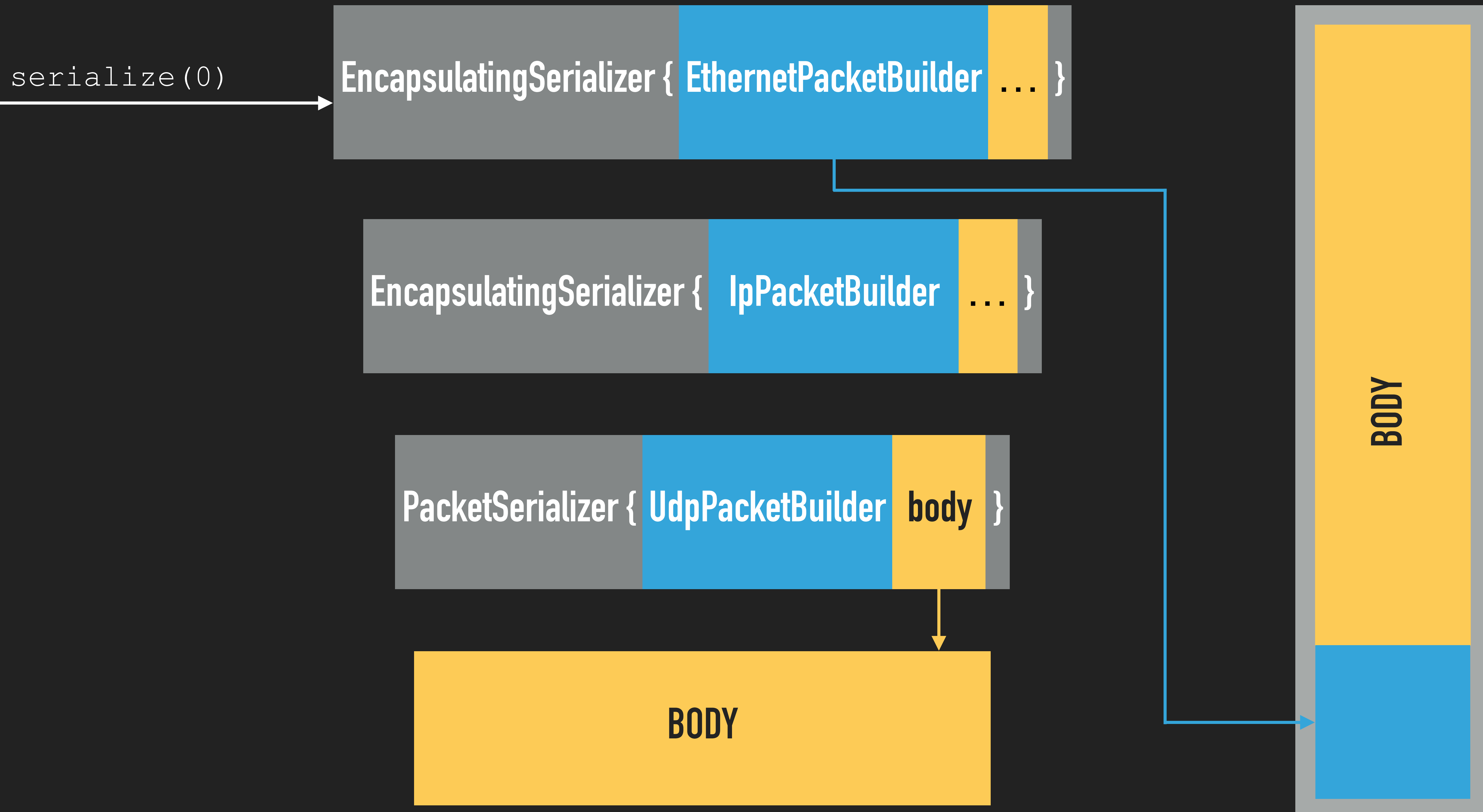
# THE SERIALIZER TRAIT

serialize(0)

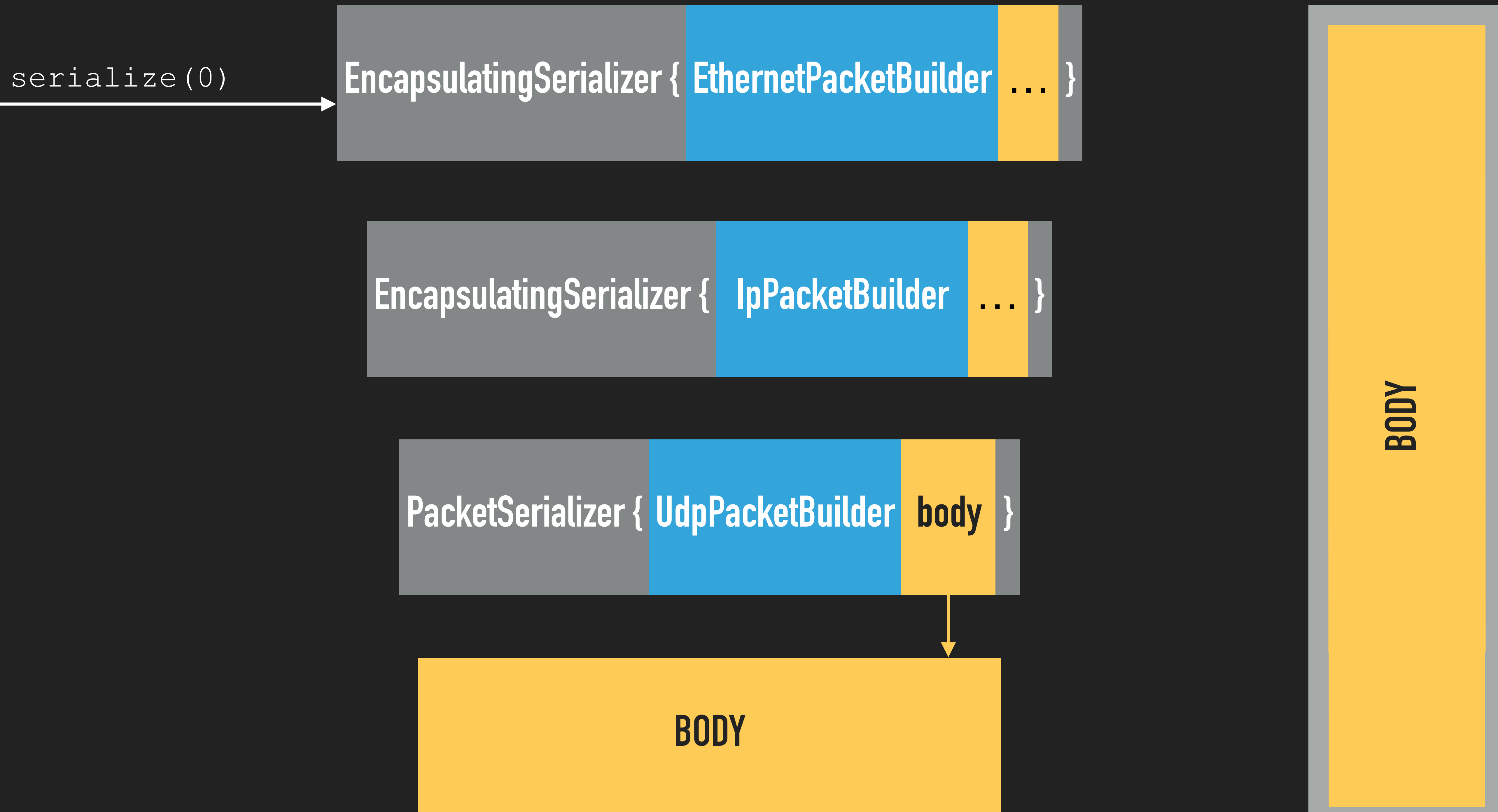




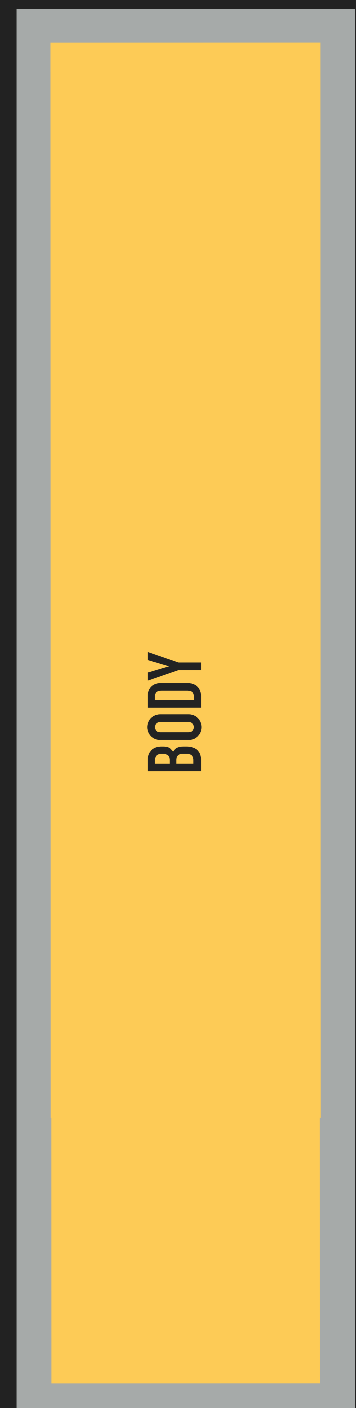
# THE SERIALIZER TRAIT



# THE SERIALIZER TRAIT



# THE SERIALIZER TRAIT



PART 3

# FORWARDING

# GOAL

- Allocate a buffer on the stack, receive an Ethernet packet
- Parse the Ethernet packet
- Parse the IP packet
- Decide to forward the packet
- Update the IP header
- Serialize the Ethernet header

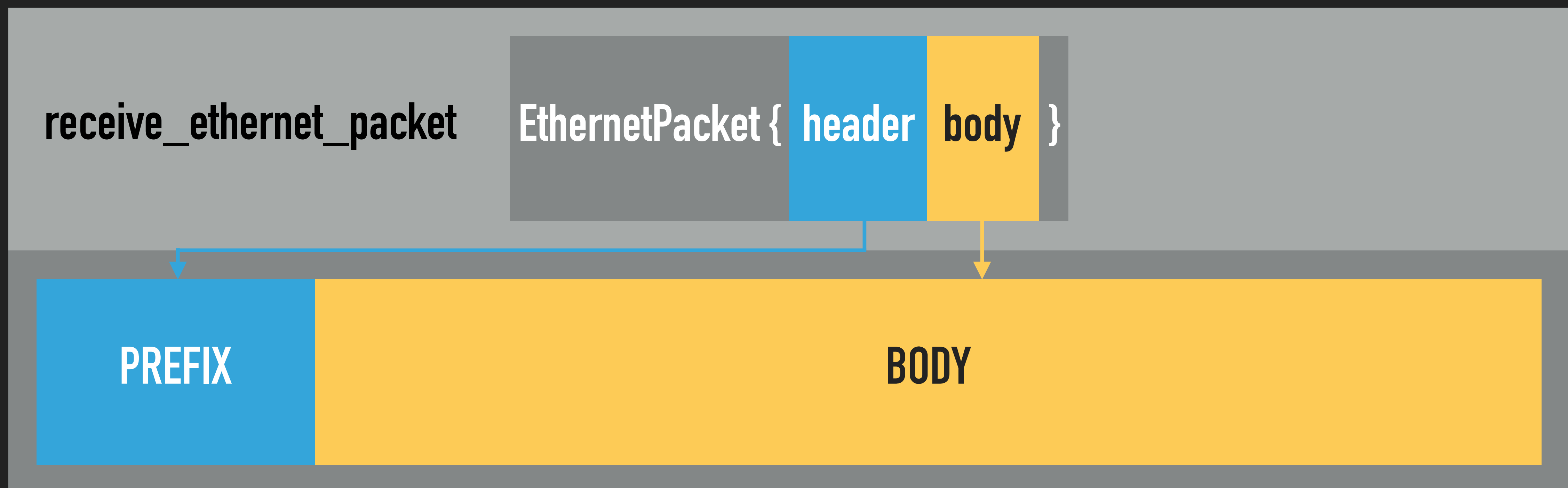
# PARSING FROM A BUFFER

# PARSING FROM A BUFFER

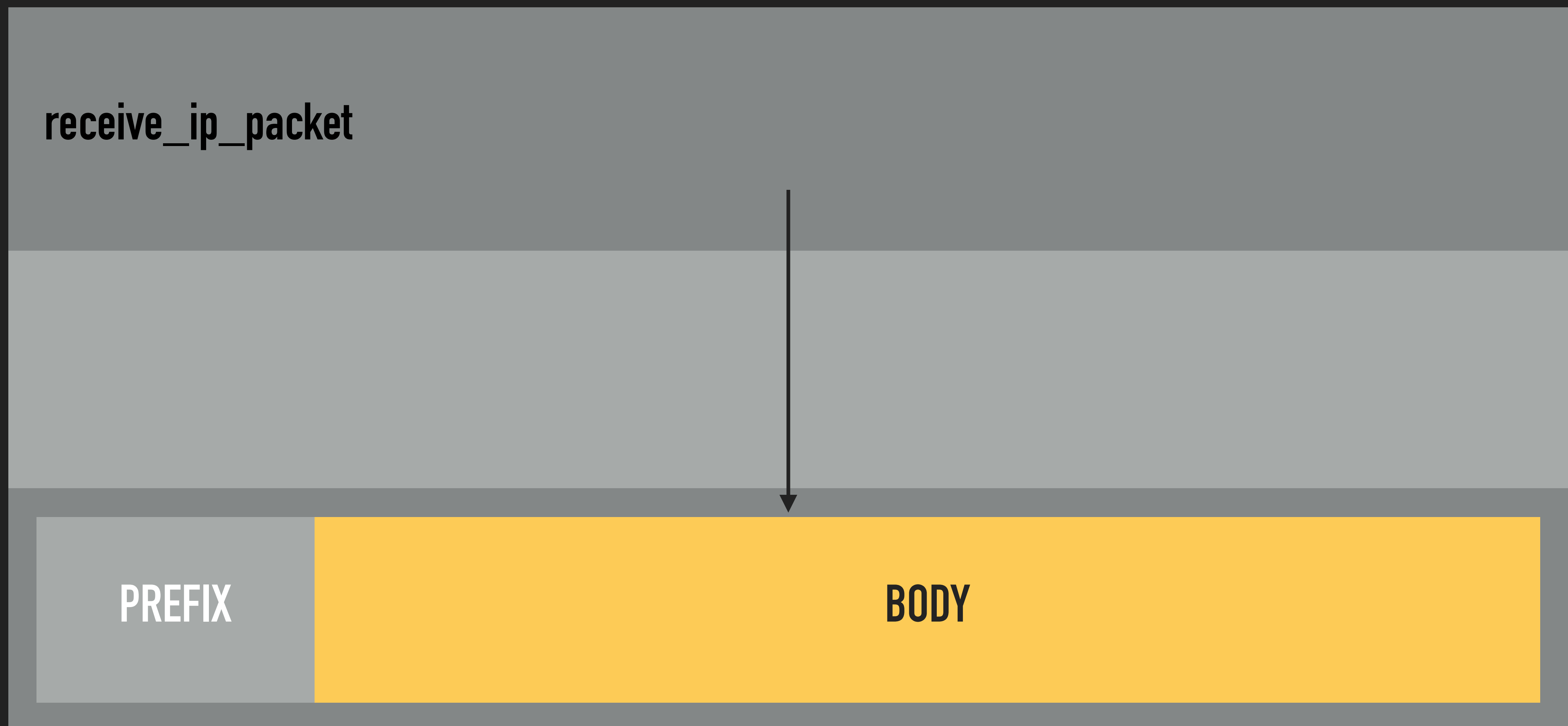
receive_ethernet_packet

BUFFER

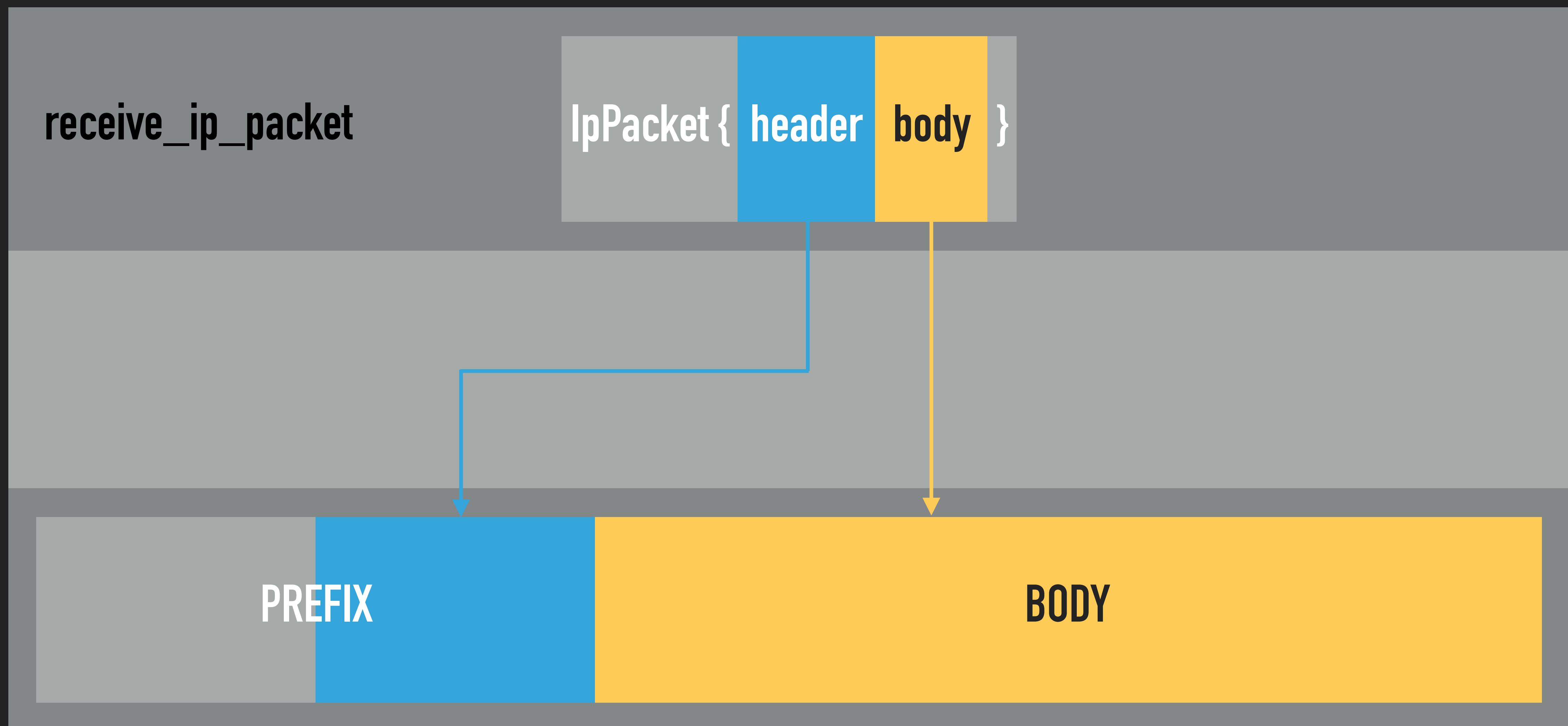
# PARSING FROM A BUFFER



# PARSING FROM A BUFFER



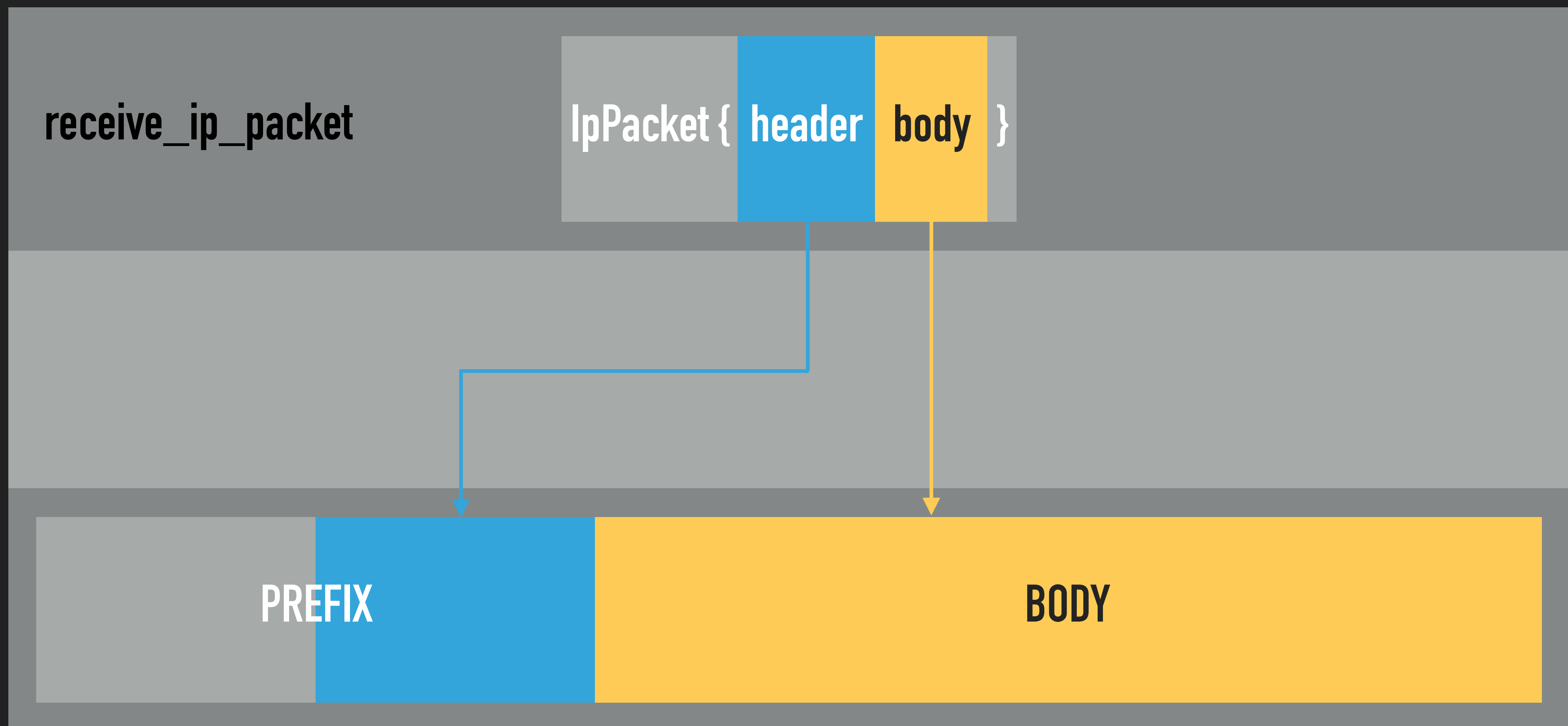
# PARSING FROM A BUFFER



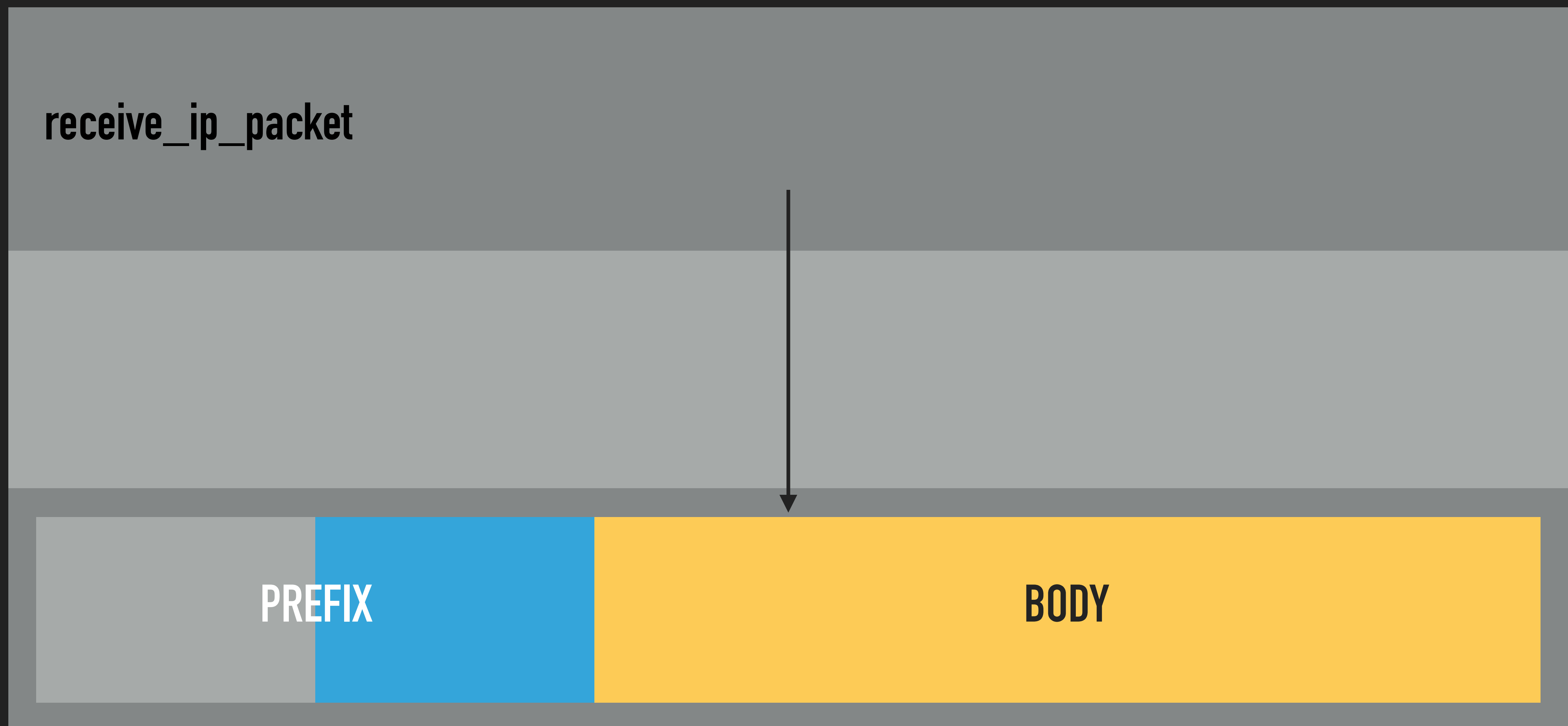
# MODIFYING IN PLACE

```
fn receive_ip_packet<B: Buffer>(mut buffer: B) {
    let packet = buffer.parse::<IpPacket>()?;
    if forward(&packet) {
        packet.decrement_ttl();
    }
}
```

# MODIFYING IN PLACE



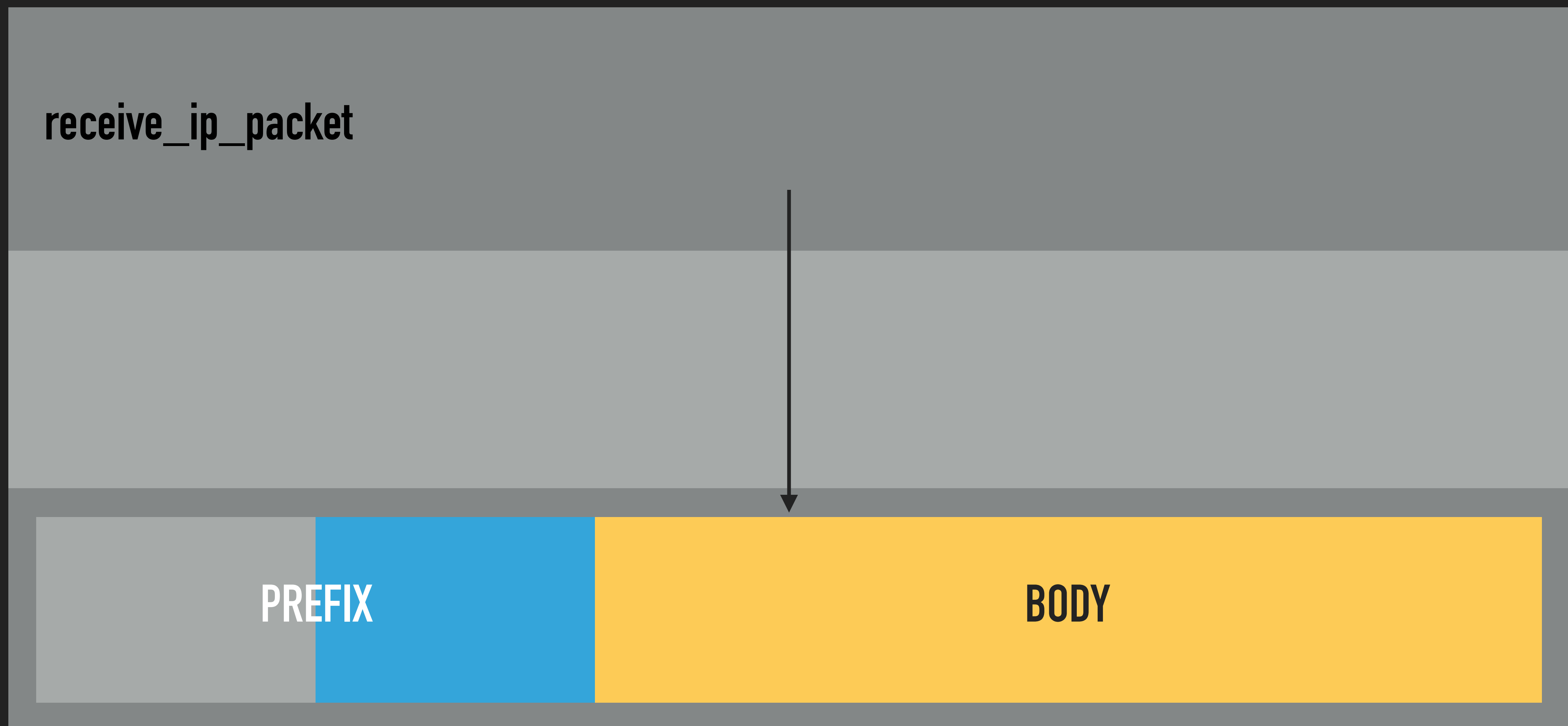
# MODIFYING IN PLACE



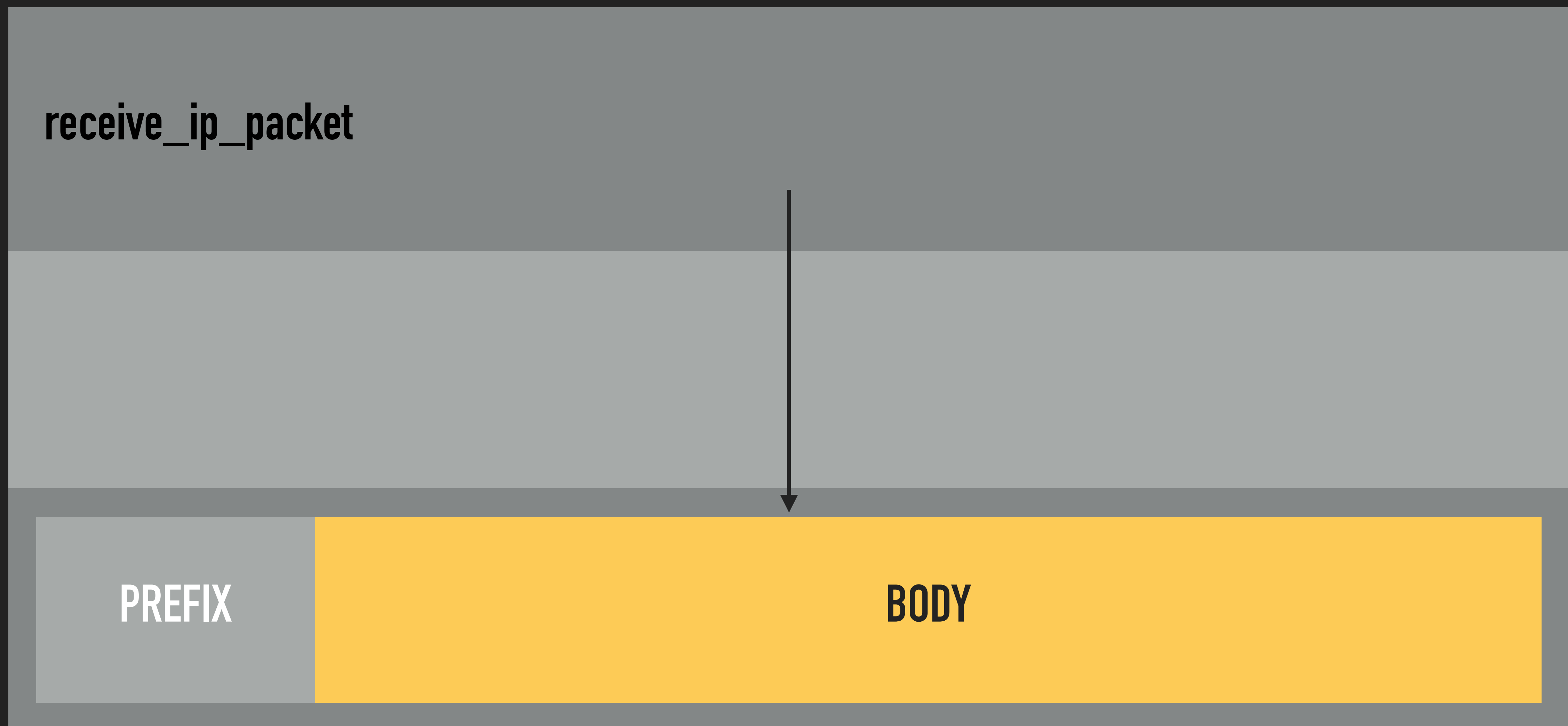
# SERIALIZING FROM A BUFFER

```
fn receive_ip_packet<B: Buffer>(mut buffer: B) {
    let packet = buffer.parse::<IpPacket>()?;
    if forward(&packet) {
        packet.decrement_ttl();
        let header_len = packet.header_len();
        buffer.undo_parse(header_len);
    }
}
```

# SERIALIZING FROM A BUFFER



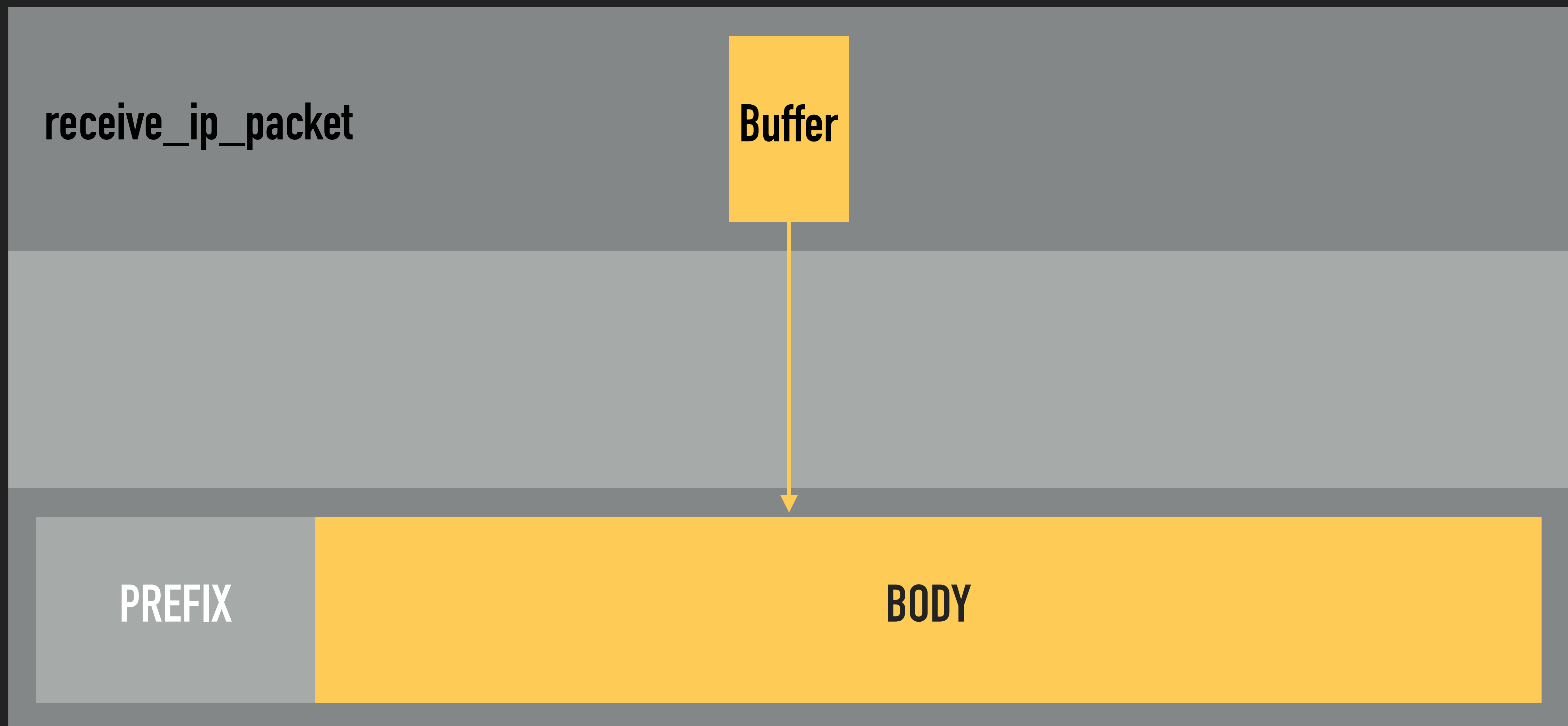
# SERIALIZING FROM A BUFFER



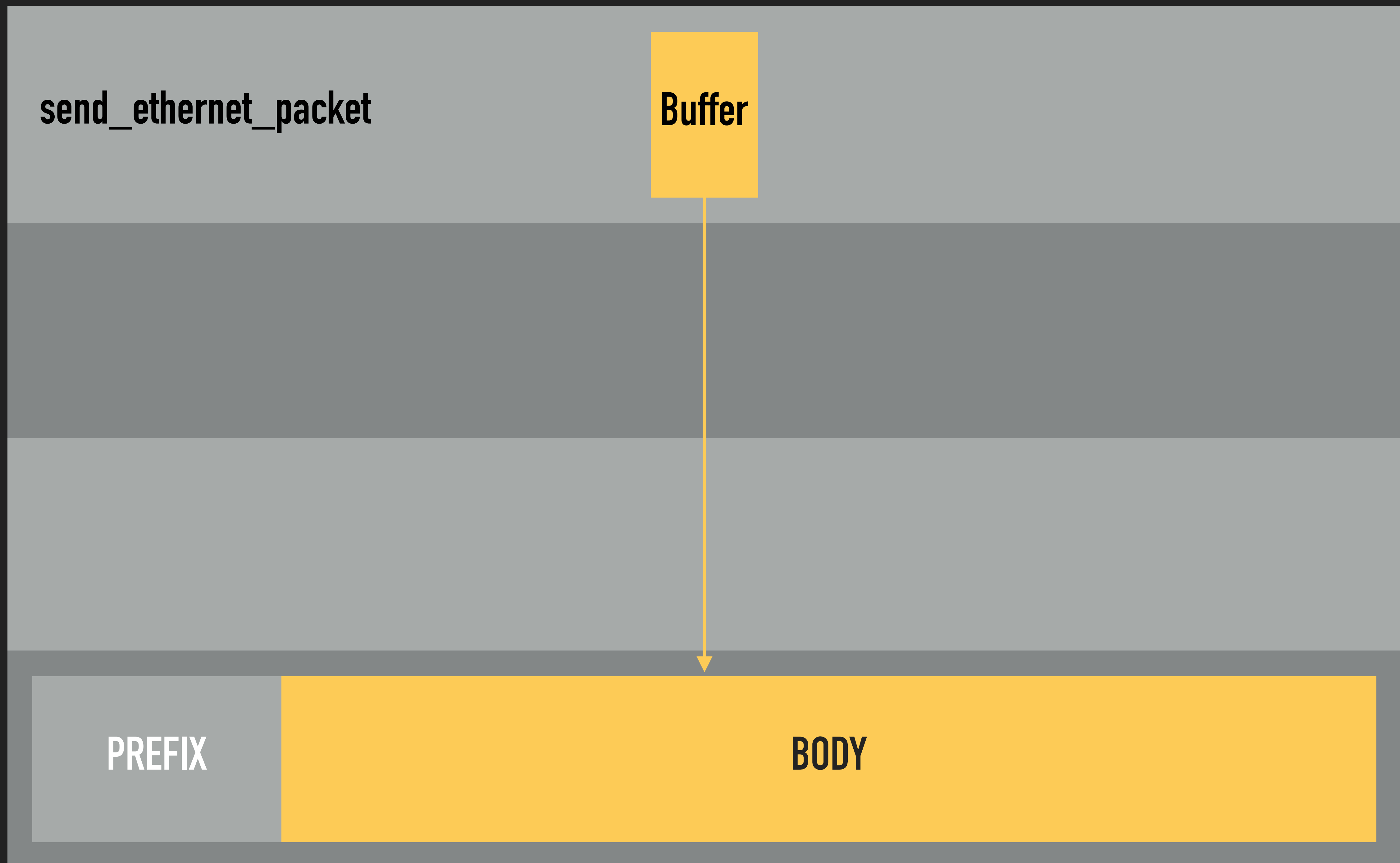
# SERIALIZING FROM A BUFFER

```
fn receive_ip_packet<B: Buffer>(mut buffer: B) {
    let packet = buffer.parse::<IpPacket>()?;
    if forward(&packet) {
        packet.decrement_ttl();
        let header_len = packet.header_len();
        buffer.undo_parse(header_len);
        ethernet::send_ethernet_packet(buffer);
    }
}
```

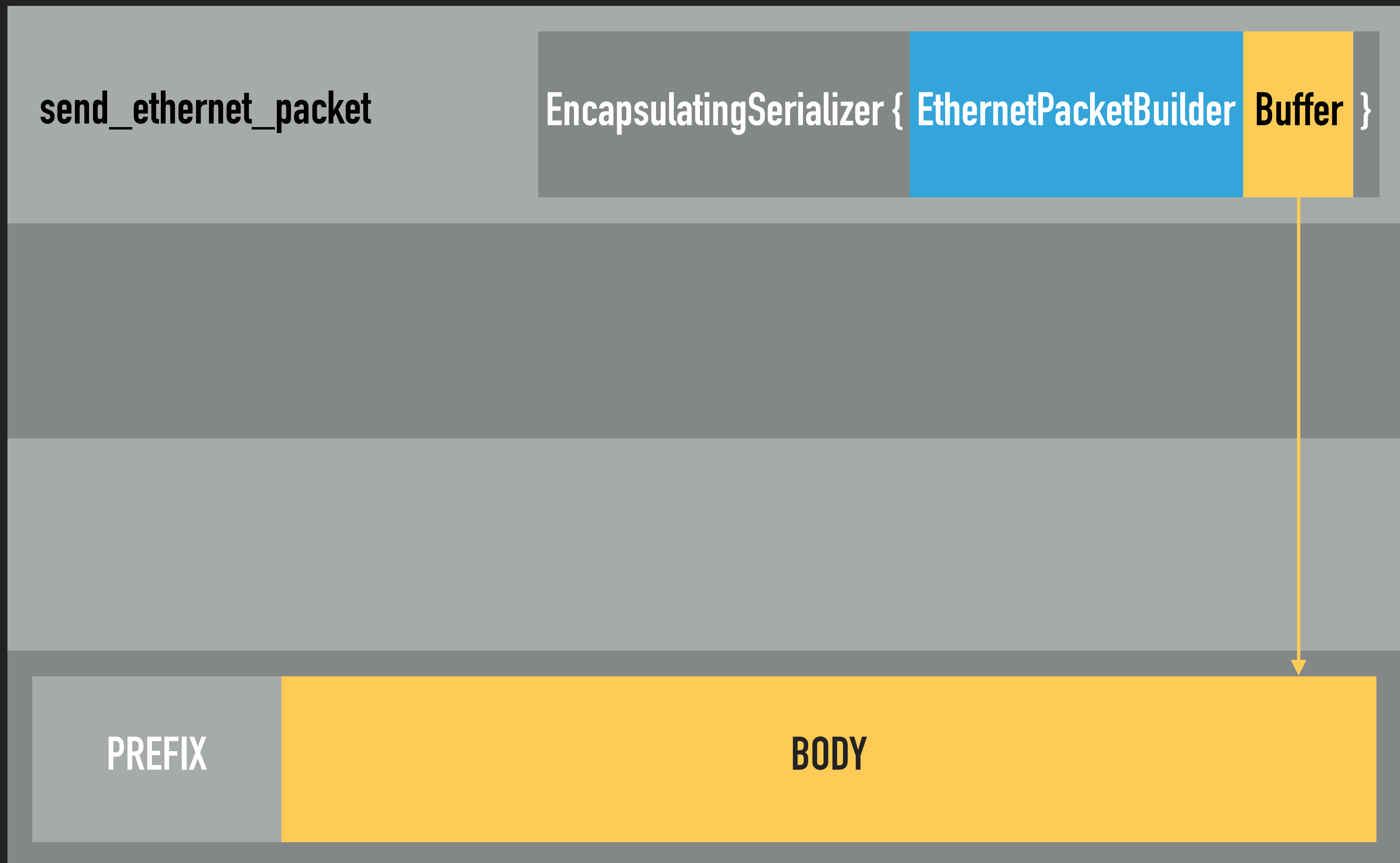
# SERIALIZING FROM A BUFFER



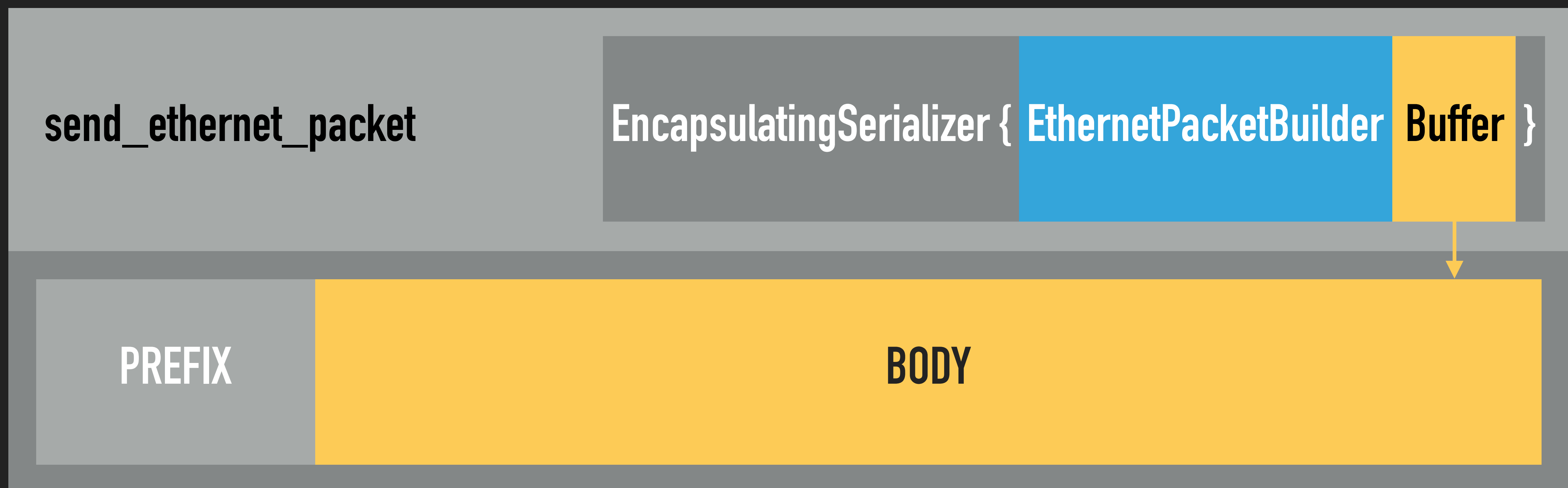
# SERIALIZING FROM A BUFFER



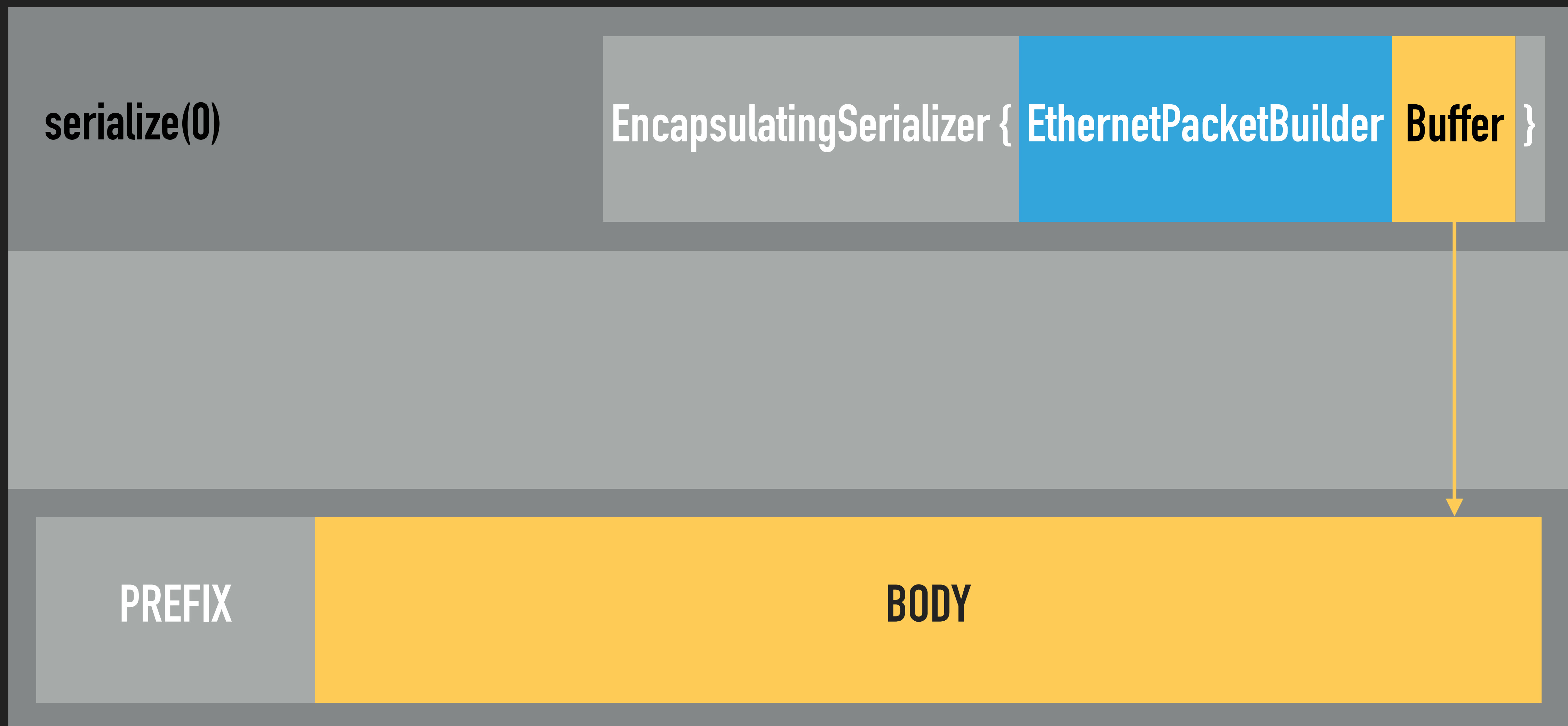
# SERIALIZING FROM A BUFFER



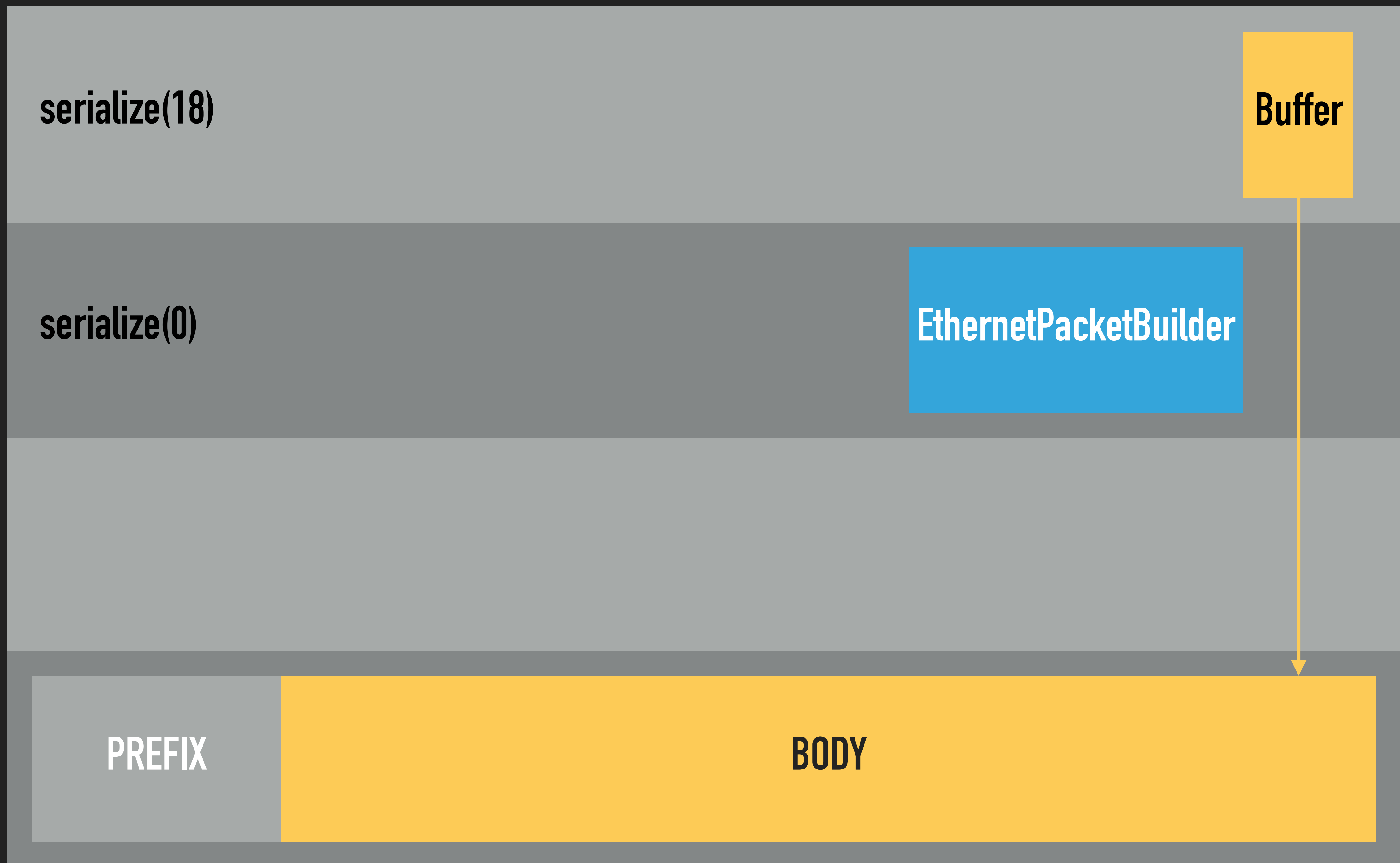
# SERIALIZING FROM A BUFFER



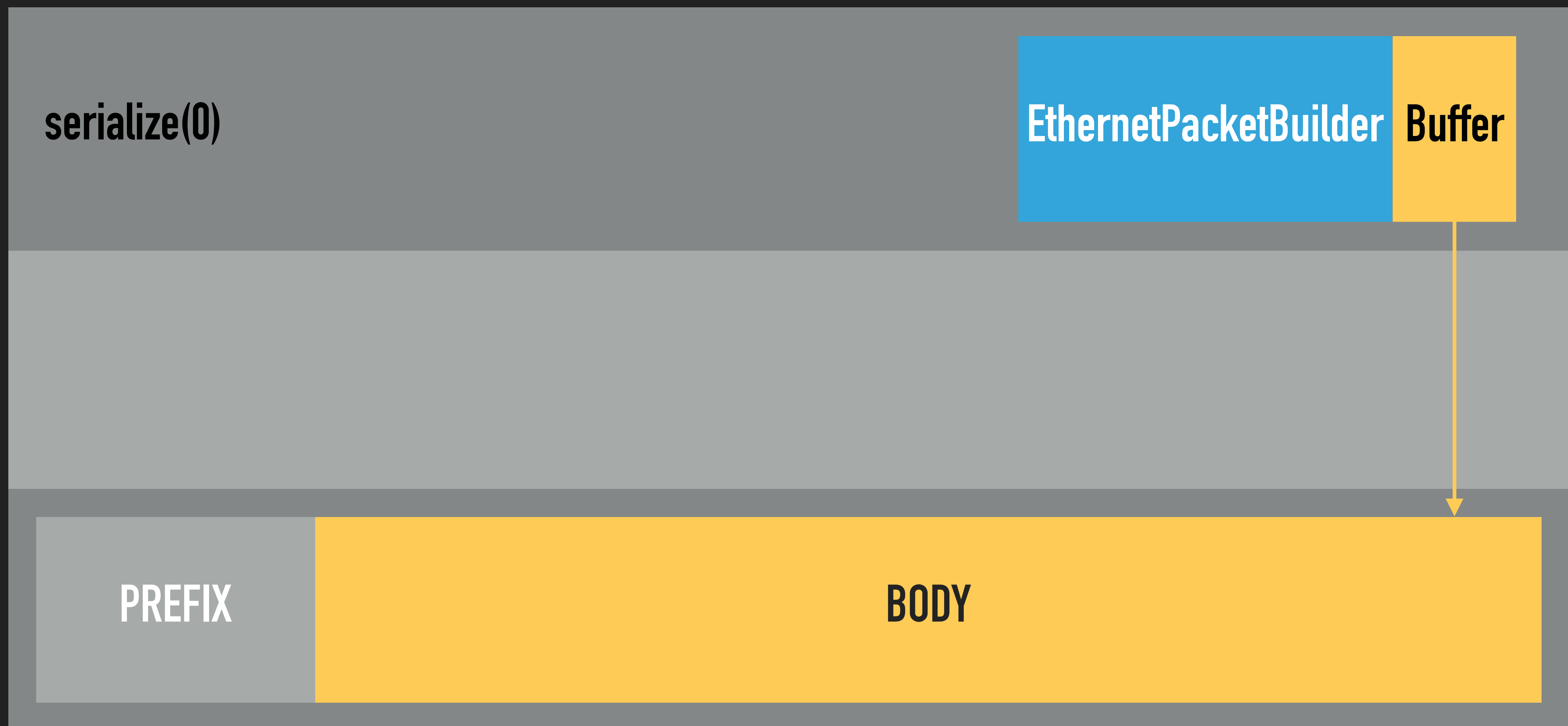
# SERIALIZING FROM A BUFFER



# SERIALIZING FROM A BUFFER

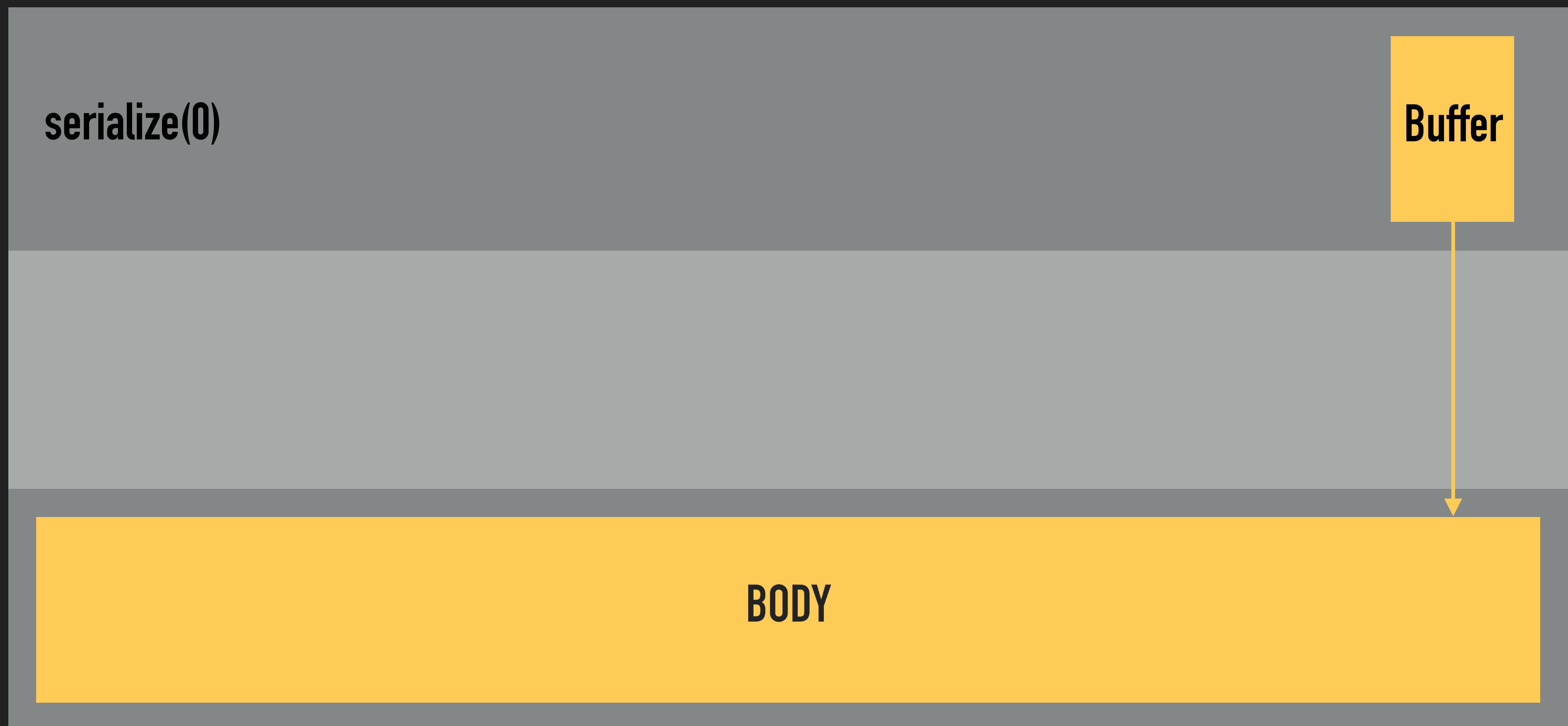


# SERIALIZING FROM A BUFFER



# SERIALIZING FROM A BUFFER

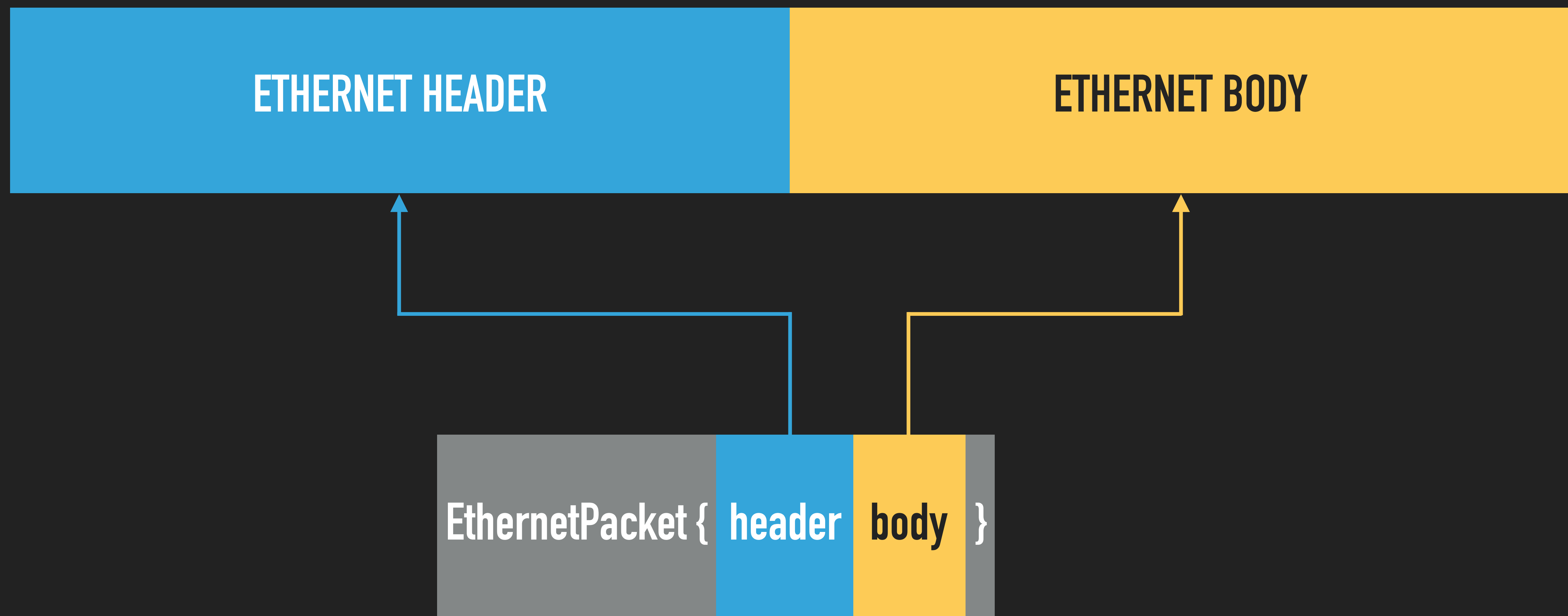
# SERIALIZING FROM A BUFFER



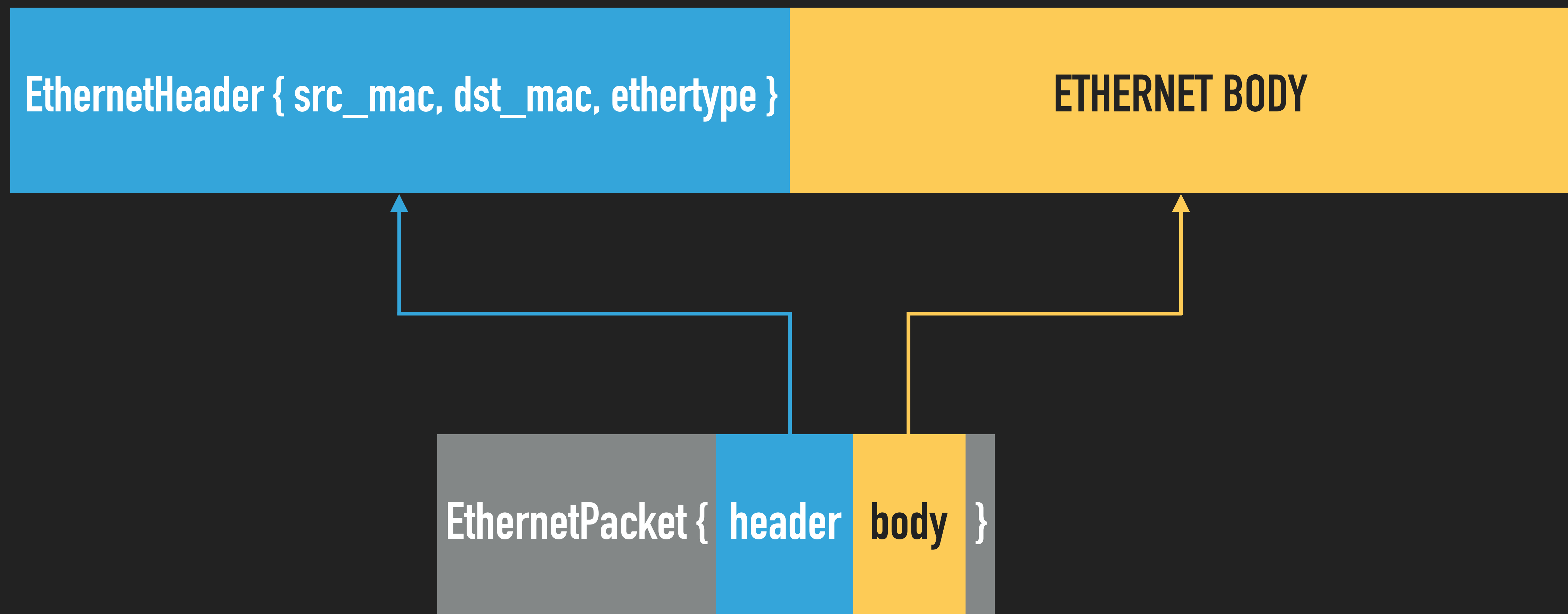
PART 4

# ZERO-COPY

# PARSING FROM A BUFFER



# PARSING FROM A BUFFER



# FROMBYTES

```
unsafe trait FromBytes {}
```

# FROMBYTES

- Any size_of<T>() bytes is a valid T
- Structs, arrays, unions: All fields/elements are FromBytes
- Enums: C-like, with $2^{bits}$ variants

# FROMBYTES

```
#[derive(FromBytes)]
struct EthernetHeader {
    dst_mac: [u8; 6],
    src_mac: [u8; 6],
    ethertype: [u8; 2],
}
```

# UNALIGNED

```
unsafe trait Unaligned {}
```

# UNALIGNED

```
#[derive(FromBytes, Unaligned)]
struct EthernetHeader {
    dst_mac: [u8; 6],
    src_mac: [u8; 6],
    ethertype: [u8; 2],
}
```

# FROMBYTES + UNALIGNED

```
fn ref_from_bytes<T>(bytes: &[u8]) -> Option<&T>
where
    T: FromBytes + Unaligned,
{ ... }
```

# FROMBYTES + UNALIGNED

```
fn parse<B: Buffer>(buf: &mut B) -> Result<Self, Err> {
    let header = buf.take_obj<T>()?;
    EthernetPacket {
        header,
        body: buf.body(),
    }
}
```

## CONCLUSION

- In C, C++, this code can be written, but *very unsafely*
- Sometimes, it's so unsafe, it can't be done
  - Parallel CSS layout in Chrome vs Firefox
- Safety brings speed, developer friendliness

# THANKS!

- hello@joshlf.com
- fuchsia.googlesource.com/garnet/+/master/…
  - bin/recovery_netstack/core
  - public/rust/zerocopy
  - public/rust/packet
    - fuchsia-review.googlesource.com/c/garnet/+/…
      - 210748, 210749, 210750