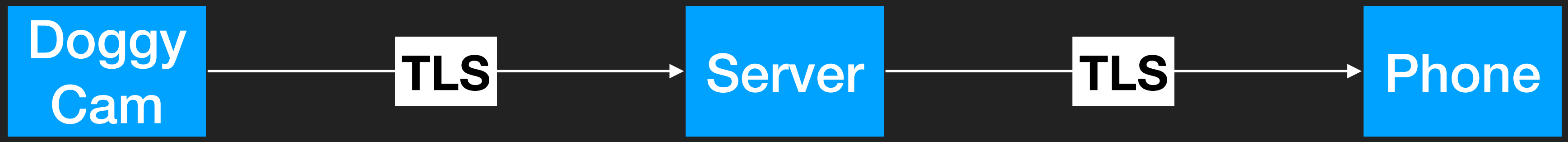
THE C TRANSPORT IMPLEMENTATION

# OCKAM
# SECURE COMMUNICATIONS STACK

# Why Ockam Channels

## Or…why NOT just TLS?



- TLS is not routable. Server decrypts, then encrypts packet (not secure over multiple hops)
    - Opens window of vulnerability
        - Unscrupulous employees
        - Hackers
        - Unauthorized data mining
- Ockam Channels are simpler
    - Less resource-intensive
    - No PKI to manage
- It not either/or: Ockam Channels are application-level and can operate over TLS

# WHAT IS A CHANNEL

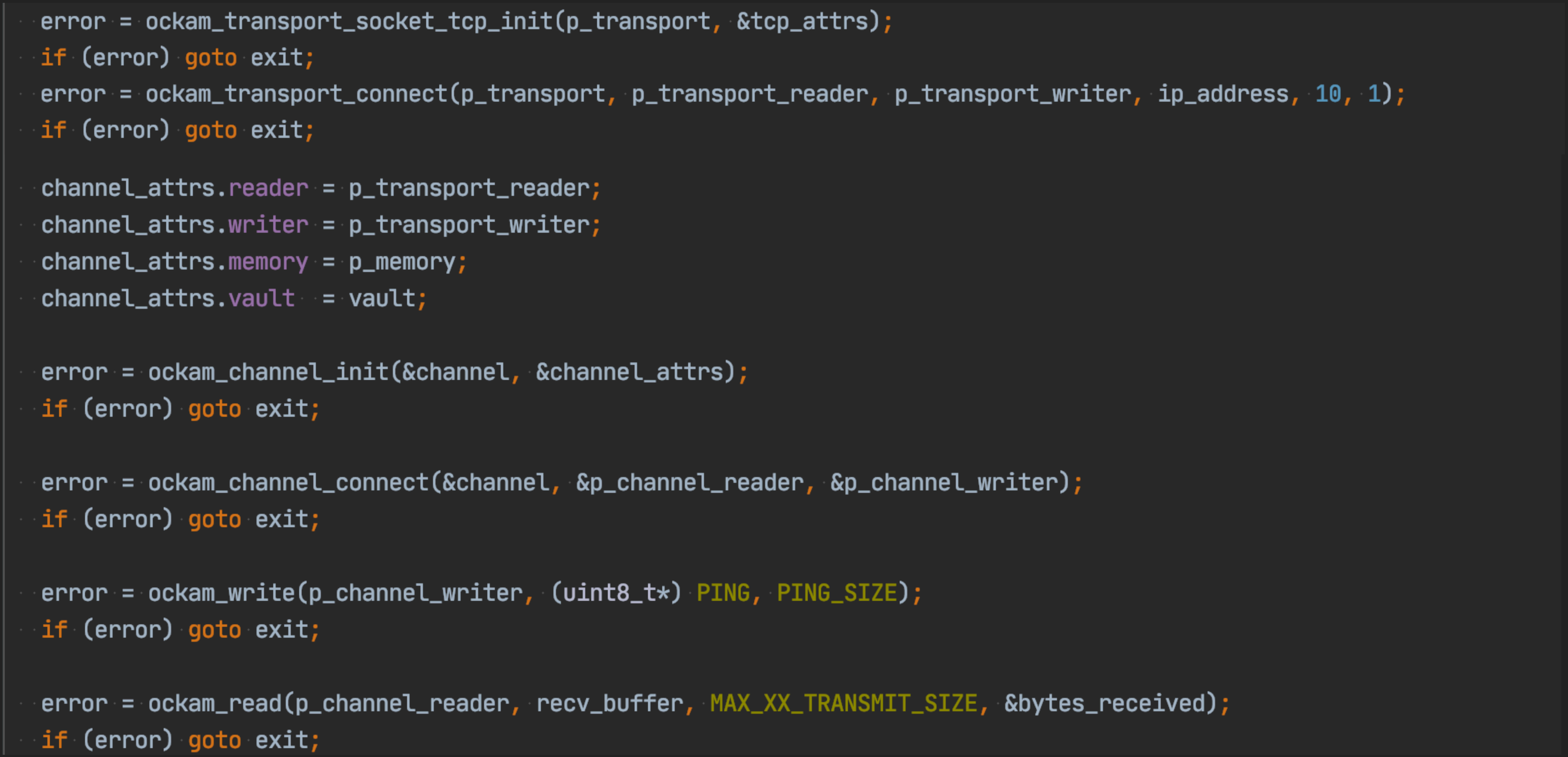
- A Channel is a way to privately send secure messages across a network of 1 to n hops
  - Private: ultimate destination is hidden
  - Secure: message is encrypted
- A Channel is composed of:
  - Transport
    - UDP, TCP sockets currently implemented
  - Key exchange protocol
    - XX pattern of the noise protocol framework currently implemented
  - Routing information
    - Elixir implementation routes across multiple Ockam servers
    - Routing not currently support in C (endpoint implementation only)

# OCKAM INTERFACES

# ABSTRACTING THE IMPLEMENTATION

- Each module has a public interface that is unchanged regardless of underlying implementation
- The specific implementation is selected by the initialization function
- Except for initialization, the APIs are the same across implementations
    - For example, to switch from UDP to TCP, just change the initialization call. That’s it.

# SNEAK PREVIEW



```c
  error = ockam_transport_socket_tcp_init(p_transport, &tcp_attrs);
  if (error) goto exit;
  error = ockam_transport_connect(p_transport, p_transport_reader, p_transport_writer, ip_address, 10, 1);
  if (error) goto exit;

  channel_attrs.reader = p_transport_reader;
  channel_attrs.writer = p_transport_writer;
  channel_attrs.memory = p_memory;
  channel_attrs.vault  = vault;

  error = ockam_channel_init(&channel, &channel_attrs);
  if (error) goto exit;

  error = ockam_channel_connect(&channel, &p_channel_reader, &p_channel_writer);
  if (error) goto exit;

  error = ockam_write(p_channel_writer, (uint8_t*) PING, PING_SIZE);
  if (error) goto exit;

  error = ockam_read(p_channel_reader, recv_buffer, MAX_XX_TRANSMIT_SIZE, &bytes_received);
  if (error) goto exit;
```

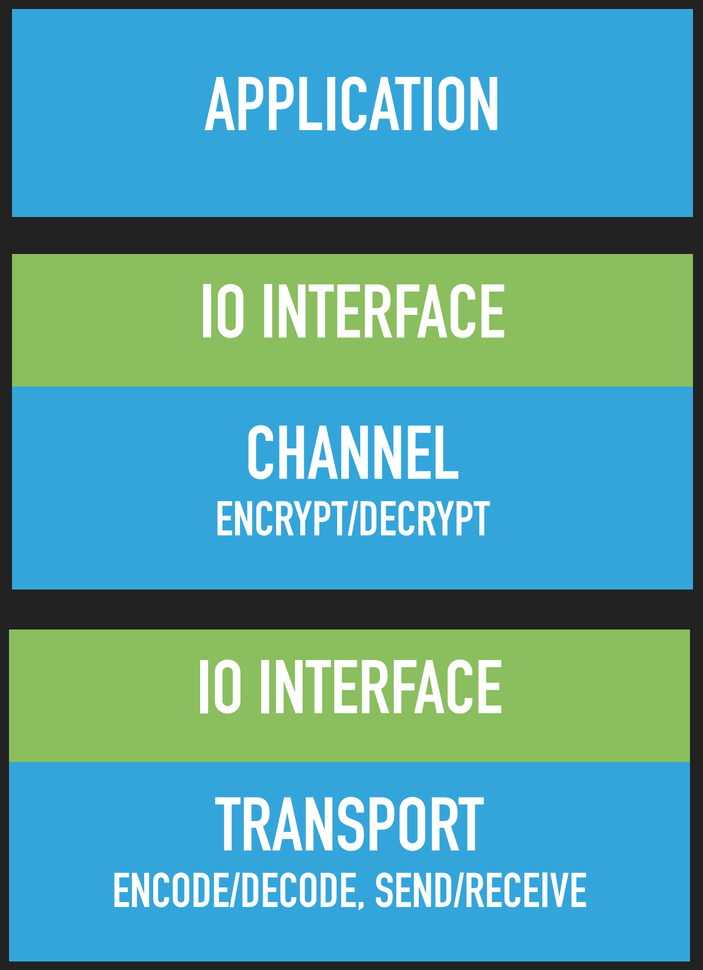
# OVERVIEW OF MODULES

- IO: for performing reads/writes over any transport or channel
  - What it does: Defines IO interface for Transport and Channel
    - Similar to posix file descriptors
  - Interface only
- Transport
  - What it does: Performs network IO
  - Implements: IO interface
- Key Agreement
  - What it does: Performs key exchange protocol  and performs encryption/decryption
  - Implements: Key interface
  - Takes: IO implementation (Transport or Channel), Vault implementation

# ….OVERVIEW OF MODULES CONT'D….

- Channel
  - What it does: performs secure & private IO using
  - Implements: IO interface
  - Takes: IO implementation
- Other:
  - Memory: pluggable memory allocator
  - Vault: abstraction over TPMs, HSM, secure enclaves, etc.

# TRANSPORT STACK



APPLICATION

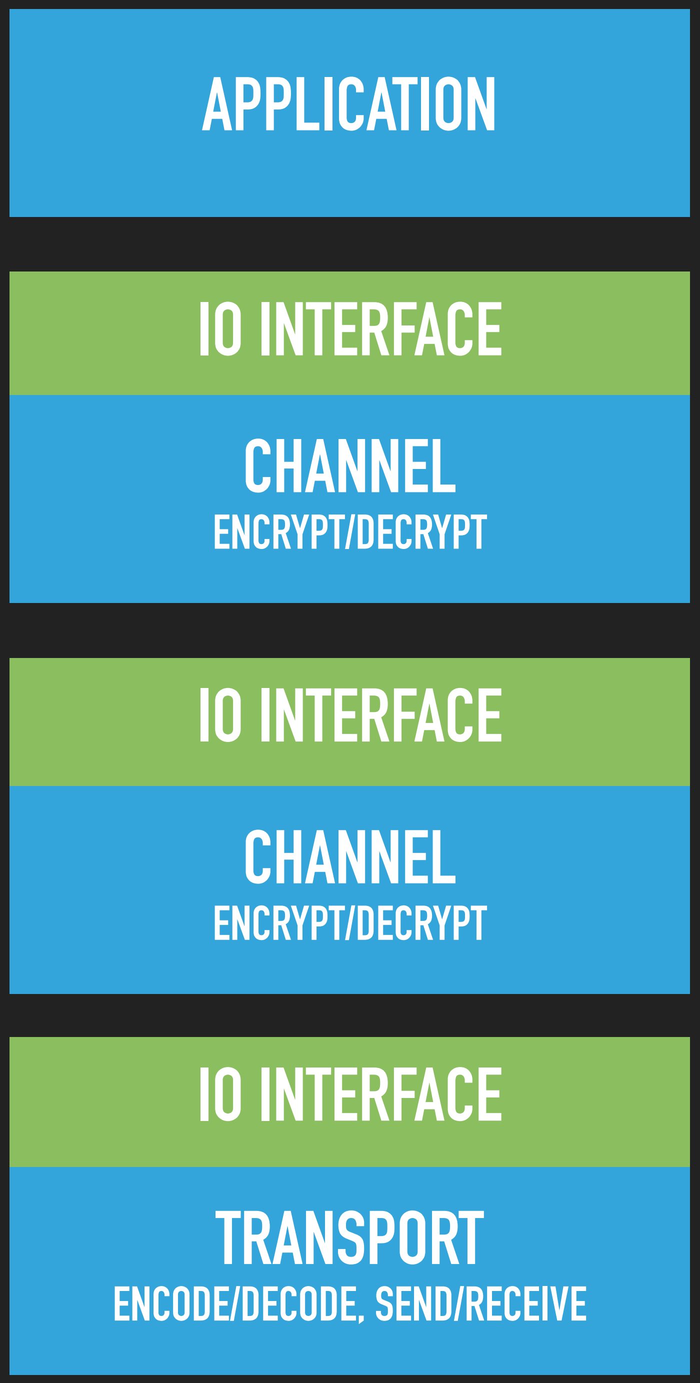
IO INTERFACE

CHANNEL
ENCRYPT/DECRYPT

IO INTERFACE

TRANSPORT
ENCODE/DECODE, SEND/RECEIVE

Base Case



APPLICATION

IO INTERFACE

CHANNEL
ENCRYPT/DECRYPT

IO INTERFACE

CHANNEL
ENCRYPT/DECRYPT

IO INTERFACE

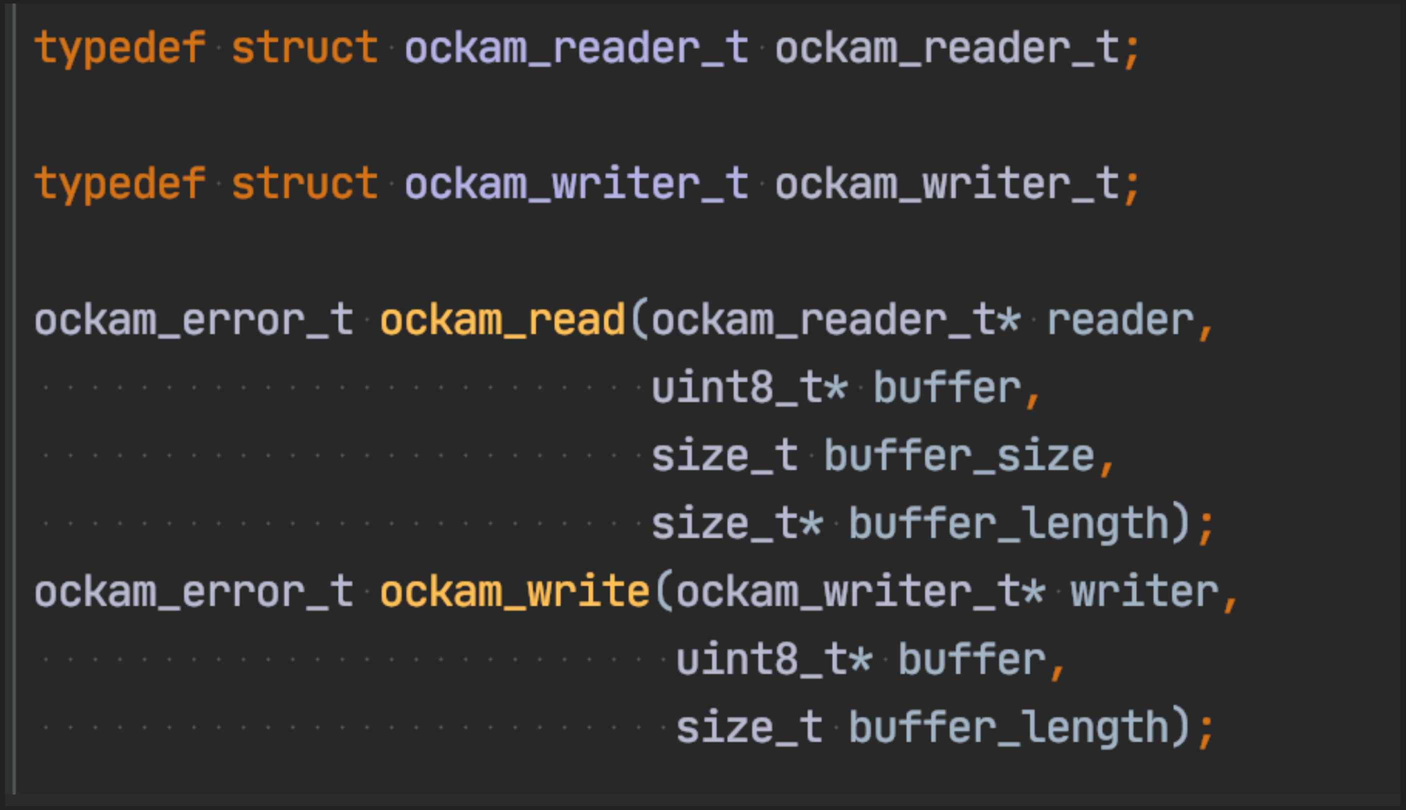
TRANSPORT
ENCODE/DECODE, SEND/RECEIVE

Tunneled Channel

# 10

# IO INTERFACE

All modules capable of performing IO implement the ockam_io interface:



```c
typedef struct ockam_reader_t ockam_reader_t;

typedef struct ockam_writer_t ockam_writer_t;

ockam_error_t ockam_read(ockam_reader_t* reader,
                         uint8_t* buffer,
                         size_t buffer_size,
                         size_t* buffer_length);
ockam_error_t ockam_write(ockam_writer_t* writer,
                          uint8_t* buffer,
                          size_t buffer_length);
```

Modules that currently export IO interface: Transport, Channel

# IO IMPLEMENTATION

The interface is a pass-through to the implementation, with context passed as ockam_reader_t:

```
struct ockam_reader_t {
  ockam_error_t (*read)(void*, uint8_t*, size_t, size_t*);
  void* ctx;
};


ockam_error_t ockam_read(ockam_reader_t* p_reader,
                         uint8_t* buffer, size_t buffer_size, size_t* buffer_length)
{
  ockam_error_t error;

  if (!p_reader) {
    error = IO_ERROR_INVALID_READER;
    goto exit;
  }
  error = p_reader->read(p_reader->ctx, buffer, buffer_size, buffer_length);

exit:
  if (error) log_error(error, "ockam_read");
  return error;
}
```

# TRANSPORT

## INTERFACE

- ockam_transport_xxx_init(): establishes transport type
  - This is the only implementation-specific function
  - Replace xxx with your transport of choice
- ockam_transport_connect(): prepares transport to communicate with a specified responder, returns reader/writer handles
- ockam_transport_accept(): prepares transport to communicate with an initiator, returns reader/writer handles
- ockam_read(): receive data on specified IO
- ockam_write(): write data to specified IO

# THE INITIALIZATION FUNCTION

- Implementation-specific

```
ockam_error_t ockam_transport_socket_udp_init(ockam_transport_t* p_transport,
                                              ockam_transport_socket_attributes_t* p_cfg)
```

- Takes a configuration structure with an address and memory instance

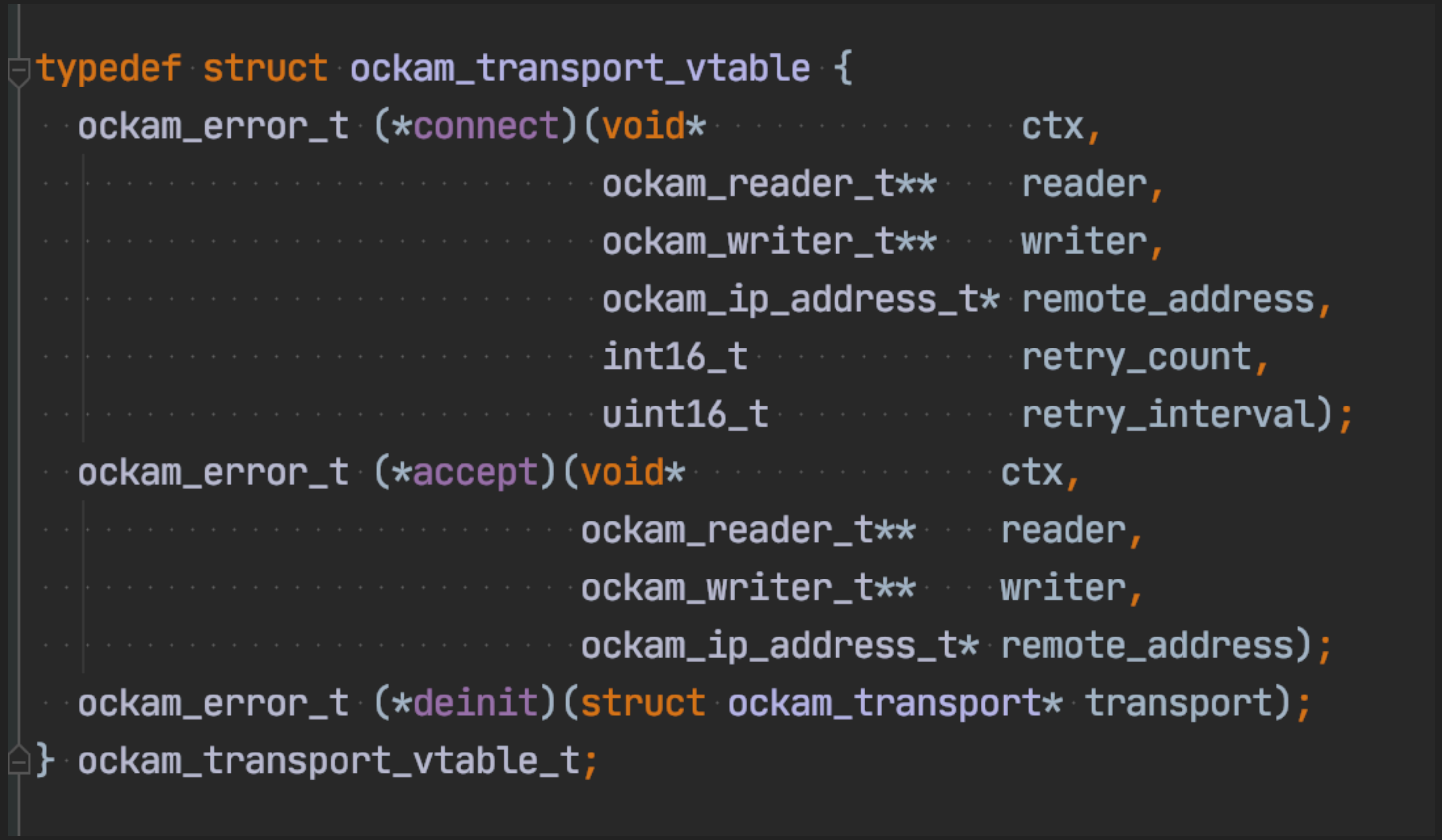
```
typedef struct ockam_ip_address_t {
  uint8_t  dns_name[MAX_DNS_NAME_LENGTH];     // "www.name.ext"
  uint8_t  ip_address[MAX_IP_ADDRESS_LENGTH]; //"xxx.xxx.xxx.xxx"
  uint16_t port;
} ockam_ip_address_t;

typedef struct ockam_transport_socket_attributes {
  ockam_ip_address_t listen_address;
  ockam_memory_t*    p_memory;
} ockam_transport_socket_attributes_t;
```

- Allocates whatever memory it needs

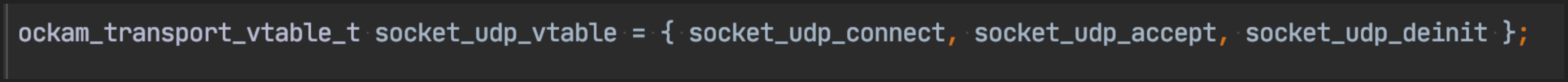
- Returns opaque handle to the transport instance

# THE INTERFACE IMPLEMENTATION

- Implements a vtable which is referenced by the implementation-specific initialization function, thus resolving the linker dependency



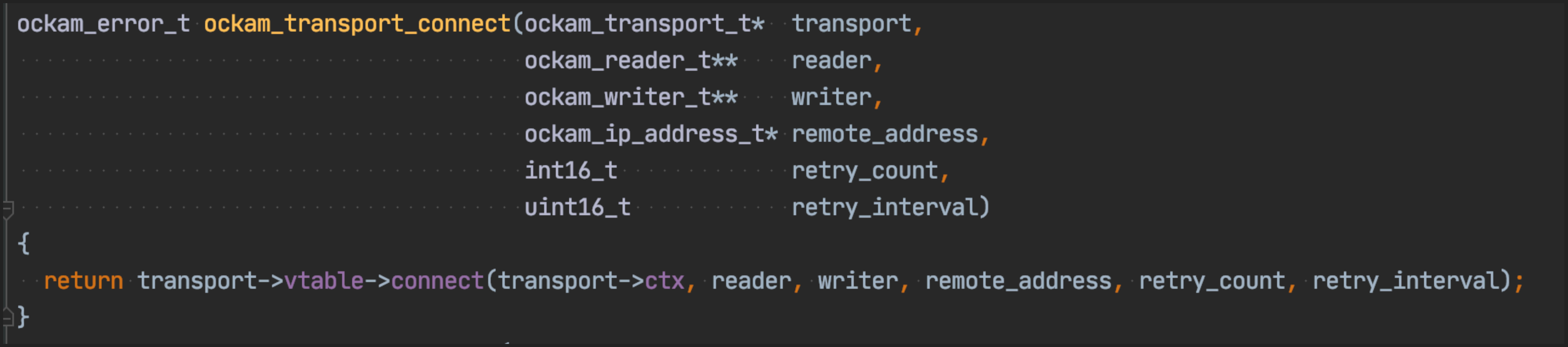
```c
typedef struct ockam_transport_vtable {
  ockam_error_t (*connect)(void*              ctx,
                           ockam_reader_t**   reader,
                           ockam_writer_t**   writer,
                           ockam_ip_address_t* remote_address,
                           int16_t            retry_count,
                           uint16_t           retry_interval);
  ockam_error_t (*accept)(void*              ctx,
                          ockam_reader_t**   reader,
                          ockam_writer_t**   writer,
                          ockam_ip_address_t* remote_address);
  ockam_error_t (*deinit)(struct ockam_transport* transport);
} ockam_transport_vtable_t;
```



```c
ockam_transport_vtable_t socket_udp_vtable = { socket_udp_connect, socket_udp_accept, socket_udp_deinit };
```

# CONNECT - INITIATOR

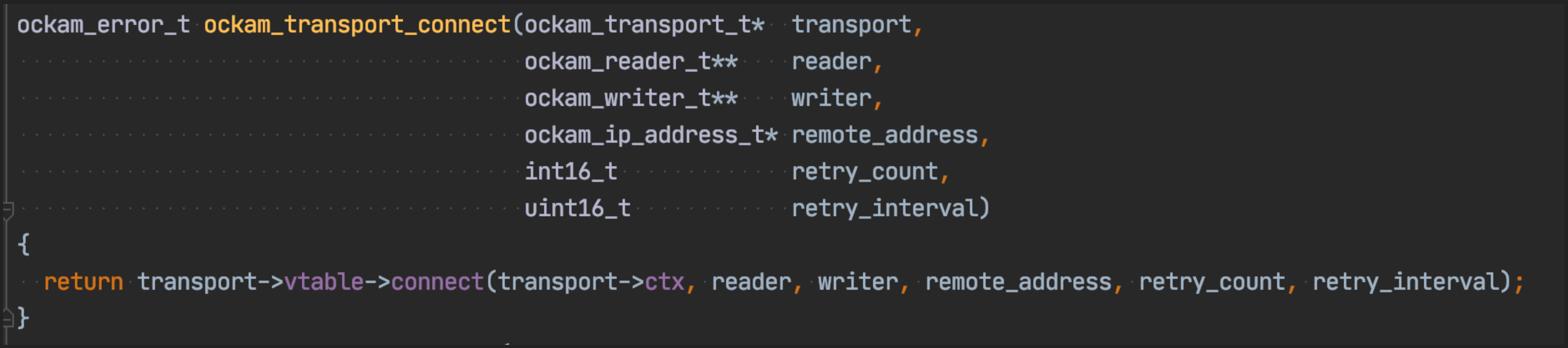
- Takes remote address, retry count, retry interval
- Returns reader/writer handles
- Opens socket
- TCP only: establishes connection to remote address
- UDP: saves remote address



```
ockam_error_t ockam_transport_connect(ockam_transport_t*  transport,
                                      ockam_reader_t**    reader,
                                      ockam_writer_t**    writer,
                                      ockam_ip_address_t* remote_address,
                                      int16_t             retry_count,
                                      uint16_t            retry_interval)
{
  return transport->vtable->connect(transport->ctx, reader, writer, remote_address, retry_count, retry_interval);
}
```

# ACCEPT - RESPONDER

- Returns reader/writer handles
- TCP:
  - Listens, blocks (for now) until connect request received and connection established
- UDP:
  - Binds socket to specified address

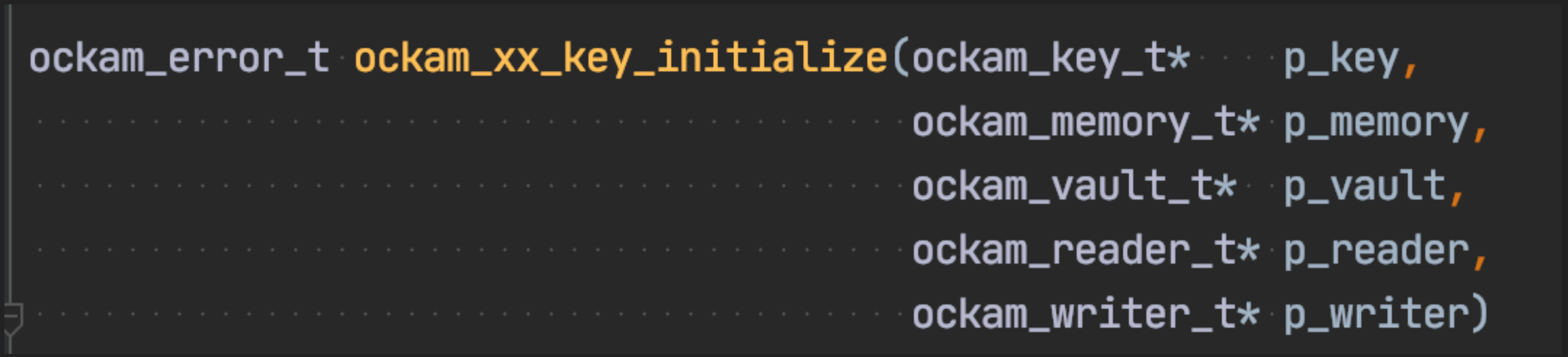


```
ockam_error_t ockam_transport_connect(ockam_transport_t*  transport,
                                      ockam_reader_t**    reader,
                                      ockam_writer_t**    writer,
                                      ockam_ip_address_t* remote_address,
                                      int16_t             retry_count,
                                      uint16_t            retry_interval)
{
  return transport->vtable->connect(transport->ctx, reader, writer, remote_address, retry_count, retry_interval);
}
```

# KEY_AGREEMENT

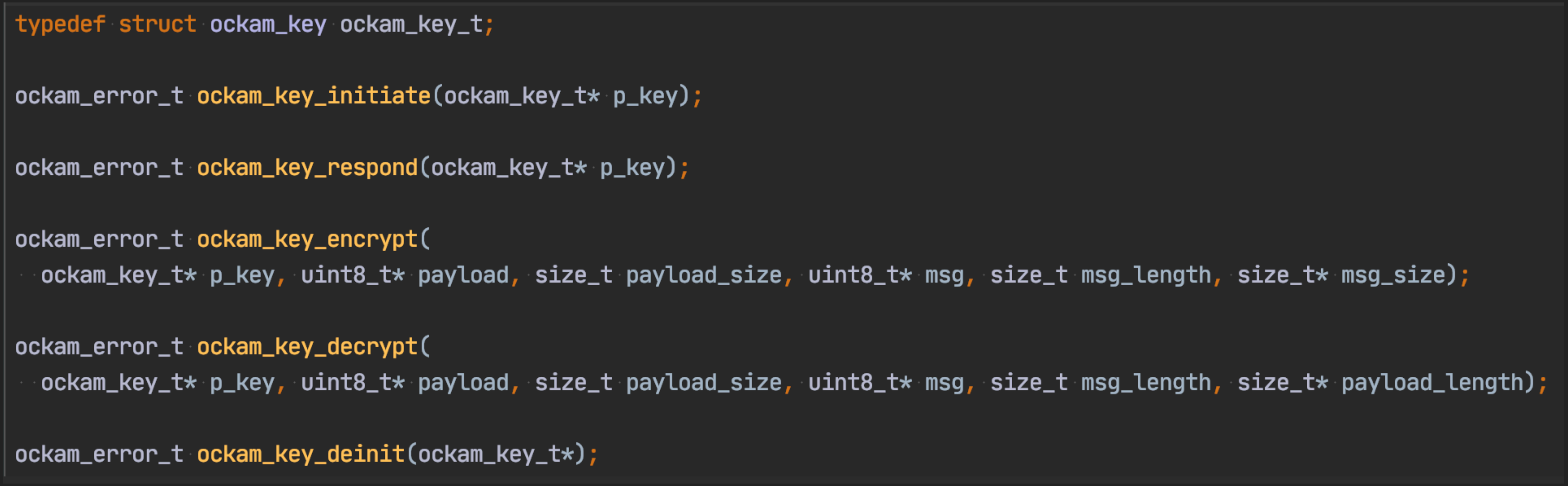
# OCKAM_XX_KEY_INITIALIZE()

- Reader/writer can be anything that exposes the ockam_io interface (i.e. channel or transport)
- This allows for “tunneling” of channels



```
ockam_error_t ockam_xx_key_initialize(ockam_key_t*    p_key,
                                      ockam_memory_t* p_memory,
                                      ockam_vault_t*  p_vault,
                                      ockam_reader_t* p_reader,
                                      ockam_writer_t* p_writer)
```

# INTERFACE

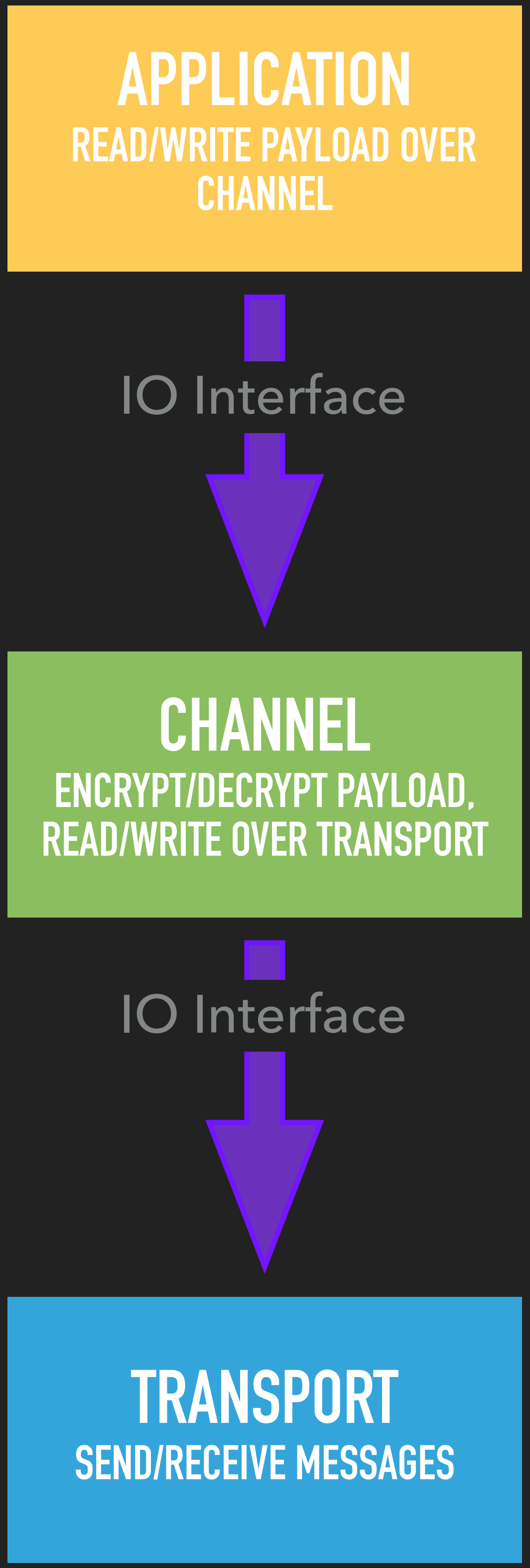


```c
typedef struct ockam_key ockam_key_t;

ockam_error_t ockam_key_initiate(ockam_key_t* p_key);

ockam_error_t ockam_key_respond(ockam_key_t* p_key);

ockam_error_t ockam_key_encrypt(
  ockam_key_t* p_key, uint8_t* payload, size_t payload_size, uint8_t* msg, size_t msg_length, size_t* msg_size);

ockam_error_t ockam_key_decrypt(
  ockam_key_t* p_key, uint8_t* payload, size_t payload_size, uint8_t* msg, size_t msg_length, size_t* payload_length);

ockam_error_t ockam_key_deinit(ockam_key_t*);
```

# CHANNEL

# BRINGING IT ALL TOGETHER

- The application initializes a transport and passes the IO interface to the Channel
- The Channel performs a key exchange using the transport interface
- The Channel returns an IO interface to the application
- The application performs secure communications using the Channel IO interface



APPLICATION
READ/WRITE PAYLOAD OVER CHANNEL

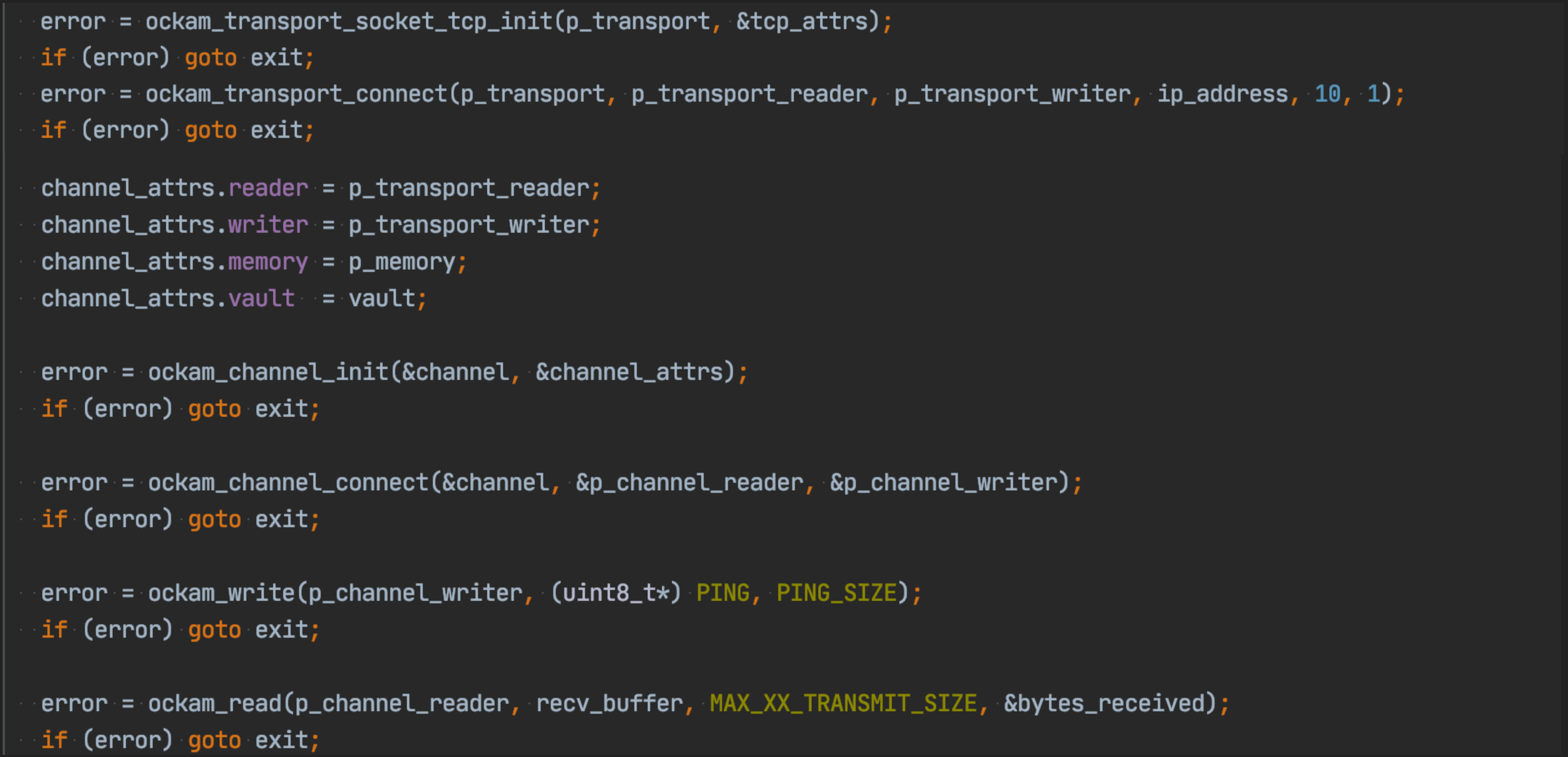
IO Interface

CHANNEL
ENCRYPT/DECRYPT PAYLOAD, READ/WRITE OVER TRANSPORT

IO Interface

TRANSPORT
SEND/RECEIVE MESSAGES

# WHAT IT LOOKS LIKE IRL



```
  error = ockam_transport_socket_tcp_init(p_transport, &tcp_attrs);
  if (error) goto exit;
  error = ockam_transport_connect(p_transport, p_transport_reader, p_transport_writer, ip_address, 10, 1);
  if (error) goto exit;

  channel_attrs.reader = p_transport_reader;
  channel_attrs.writer = p_transport_writer;
  channel_attrs.memory = p_memory;
  channel_attrs.vault  = vault;

  error = ockam_channel_init(&channel, &channel_attrs);
  if (error) goto exit;

  error = ockam_channel_connect(&channel, &p_channel_reader, &p_channel_writer);
  if (error) goto exit;

  error = ockam_write(p_channel_writer, (uint8_t*) PING, PING_SIZE);
  if (error) goto exit;

  error = ockam_read(p_channel_reader, recv_buffer, MAX_XX_TRANSMIT_SIZE, &bytes_received);
  if (error) goto exit;
```

# WHERE TO GO FROM HERE?

- Current design is single-thread, blocking
- Options:
    - Single thread, non-blocking
        - Requires app to periodically relinquish cpu for transport processing
    - Multithreaded, non-blocking
        - Not very portable
- Rust…

Have a lovely day!