SDK Concepts

Transports

The transport abstraction models a communication channel between two entities using which one side can send/receive data to/from the other side.

A transport has the following attributes -

- A source
- A destination
- A method to send data to the destination
- A method to receive and process data from the destination
- A method to close the communication channel

Common transport types

Transports can be divided into types based on the communication semantics that they provide.

Datagram based transports

Usually send and receive small packets of data.

Sample interface:

```
struct DatagramTransport {
    SocketAddress src;
    SocketAddress dst;

    void send(Buffer &&packet);
    void did_recv(Buffer &&packet);
    void close();
};
```

Example: UDP transport

Stream based transports

Usually send and receive byte streams.

Sample interface:

```
struct StreamTransport {
    SocketAddress src;
    SocketAddress dst;

    void send(Buffer &&bytes);
    void did_recv(Buffer &&bytes);
    void close();
};
```

Example: TCP transport

Message based transports

Usually send and receive delimited messages.

Sample interface:

```
struct MessageTransport {
    SocketAddress src;
    SocketAddress dst;

    void send(Buffer &&message);
    void did_recv(Buffer &&message);
    void close();
};
```

Example: Length-prefix framed transport

> **Higher-order transports**
> Transports of a particular type can also be built by wrapping another type of transport. For example, StreamTransport can provide stream semantics by wrapping a base transport providing datagram semantics. See the Higher-order transports section for more info. by wrapping a base transport providing datagram semantics.

Reusable transports

We can make our transports reusable by isolating the application-specific parts and using behaviour injection to modify them based on application requirements. Behaviour injection can take any of the following forms:

Inheritance

Isolate the application-specific parts into their own functions. Subclass and override these functions as needed.

```
struct Transport {
    SocketAddress src;
    SocketAddress dst;

    void send(Buffer &&data);
    void close();

    void did_recv(Buffer &&data); // Can subclass and override
};
```

Not that flexible. Makes writing higher-order transports difficult.

Callbacks

Isolate the application-specific parts into their own callbacks. Set these callbacks as needed.

```
struct Transport {
    SocketAddress src;
    SocketAddress dst;

    void send(Buffer &&data);
    void close();

    typedef void (*RecvFunc)(Buffer &&data);
    RecvFunc did_recv; // Can set
};
```

More of a C paradigm than C++. The callbacks need to be globally and statically addressible which

imposes significant restrictions on what can be set as a callback(crucially, no normal member functions, only static ones).

Delegates

Outsource the application-specific parts to an external object. The object can be set as needed.

```
struct TransportDelegate {
    void did_recv(Buffer &&data); // Can implement custom processing
};

template<typename DelegateType>
struct Transport {
    SocketAddress src;
    SocketAddress dst;

    void send(Buffer &&data);
    void close();

    DelegateType *delegate; // Can set
};
```

We use the delegate pattern for its flexibility and usability while remaining performant. It makes it easy to define a contract between a Transport and its delegate and as a bonus, makes the design easily portable to languages which have enforced contracts(Go interfaces, Rust traits, etc).

Event notifications

Now that we have a delegate pattern in place, we can use it to notify the delegate of significant events occuring in the transport.

Sample interface:

```
struct TransportDelegate {
    void did_close(); // Notify transport close
};

template<typename DelegateType>
struct Transport {
```

```
    DelegateType *delegate;

    void close() {
        ...
        delegate->did_close();
        ...
    }
};
```

Better delegates

In the delegates that we have above, there are significant design deficiencies:

- We have no way to know which transport the delegate call is coming from
- We need a delegate per transport to have custom logic per transport
- We have no way to directly respond based on the data received

We can fix this by simply passing the transport as a parameter to the delegate functions:

```
struct TransportDelegate {
    void did_recv(Transport<TransportDelegate> &transport, Buffer &&data) {

        // Can implement custom processing based on transport attributes
        if(transport.dst == X) {
            ...
        }

        // Can directly respond
        transport.send(response_data);
    }
};

template<typename DelegateType>
struct Transport {
    DelegateType *delegate;

    void recv_cb() {
        ...
        delegate->did_recv(*this, data);
```

```
        ...
    }
};
```

Canonical transports

Based on the design choices made above, our transports(and delegates) usually look something like this:

```
struct TransportDelegate {
    // Notify data sent
    void did_send(Transport<TransportDelegate> &transport, Buffer &&data);

    // Notify data receive
    void did_recv(Transport<TransportDelegate> &transport, Buffer &&data);

    // Notify successful dial
    void did_dial(Transport<TransportDelegate> &transport);

    // Notify close
    void did_close(Transport<TransportDelegate> &transport);
};

template<typename DelegateType>
struct Transport {
    // Source
    SocketAddress src;

    // Destination
    SocketAddress dst;

    // Delegate
    DelegateType *delegate;

    // Send data to dst
    void send(Buffer &&data);

    // Close the transport
    void close();
```

```
};
```

Higher-order transports

Higher-order transports are built by wrapping another transport as a base and customizing its behaviour. Given the use of the delegate pattern, we can simply insert the higher-order transport as a delegate for the base transport and intercept all delegate calls to modify the behaviour as needed.

Example - StreamTransport

Our higher-order transports usually look something like this:

```
template<typename DelegateType, template<typename> class SomeTransport>
struct HigherOrderTransport {
    // Source
    SocketAddress src;

    // Destination
    SocketAddress dst;

    // Delegate
    DelegateType *delegate;

    // Base transport
    typedef SomeTransport<HigherOrderTransport<DelegateType, SomeTransport>>
BaseTransport;

    BaseTransport &transport;

    // Send data to dst
    // Will eventually call transport.send to actually send the data
    void send(Buffer &&data);

    // Close the transport
    // Will eventually call transport.close to actually close the transport
    void close();

    //-------- BaseTransport delegate functions below --------//
```

```
    // Intercept data sent and do custom processing if needed
    // Will eventually call delegate->did_send to notify
    void did_send(BaseTransport &transport, Buffer &&data);

    // Intercept data receive and do custom processing if needed
    // Will eventually call delegate->did_recv to notify
    void did_recv(BaseTransport &transport, Buffer &&data);

    // Intercept successful dial and do custom processing if needed
    // Will eventually call delegate->did_dial to notify
    void did_dial(BaseTransport &transport);

    // Intercept close and do custom processing if needed
    // Will eventually call delegate->did_close to notify
    void did_close(BaseTransport &transport);
};
```

Transport Factories

Transports are created by transport factories in one of two ways:

- Dialling a destination
- Listening for a dial from a destination

Sample interface:

```
struct TransportFactory {
    // Set source address
    void bind(SocketAddress &addr);

    // Start listening for incoming dials
    void listen();

    // Dial destination address
    void dial(SocketAddress &addr);
};
```

Higher-order transport factories

Higher-order transport factories complement higher-order transports by wrapping a base transport factory. For example, StreamTransportFactory produces stream transports by wrapping around a base factory which produces datagram transports.

Endianness

Endianness refers to the ordering of components in a representation of an entity. For our purposes, we use it to refer to the ordering of bytes in multi-byte integers.

Overview

Let's look at how memory allocation and storage of multi-byte integers work. For illustration, we consider 32-bit integers. When you wish to store a 32-bit integer, the program allocates 4 bytes worth of memory.

Now we have a choice on how we want to store the integer - the 4 bytes in the integer can be mapped to the 4 bytes of memory in any order giving us 24 possible endian options. As long as we use the same ordering while storing and retrieving any integer, the program can function without error.

The most popular endian systems in use today are the big endian and little endian systems.

Big endian

Big endian refers to ordering by most significant byte to least. A 32-bit integer `0x12345678` woul be stored as

```
++++++++++++++++++++++++++++++
|  0x12 |  0x34 |  0x56 |  0x78 |
++++++++++++++++++++++++++++++
```

Example systems: TCP, IBM z/Architechture

Little endian

Little endian refers to ordering by least significant byte to most. A 32-bit integer `0x12345678` woul be stored as

```
++++++++++++++++++++++++++++++
```

```
| 0x78 | 0x56 | 0x34 | 0x12 |
++++++++++++++++++++++++++++
```

Example systems: Intel/AMD x86-64, RISC-V

Why should I care?

Endianness is crucial while serializing and deserializing data transmitted over the network between systems with different orderings. Consider the following serialization and deserialization steps happening in a big and little endian system respectively:

```
// Serialization in big endian system
char buf[4]; uint32_t i = 0xff;
memcpy(buf, &i, 4);

// Transmitted through the network as | 0x00 | 0x00 | 0x00 | 0xff |

// Deserialization in little endian system
char buf[4] = { 0x00, 0x00, 0x00, 0xff }; uint32_t i;
memcpy(&i, buf, 4);

// i now contains | 0x00 | 0x00 | 0x00 | 0xff | in memory
// which is 0xff000000 in the little endian system
// Very different from 0xff!!!
```

A mismatch in endianness causes the data read to be widly different. Best practice is to fix an endianness for the transmitted data (called network ordering) and handle conversions in the clients based on their endianness (called host ordering).

PubSub

PubSub (short for Publish/Subscribe) is an abstraction for a messaging system where senders don't send the message directly to a specific receiver. Instead, messages are grouped into channels that interested receivers (also called subscribers) can subscribe to. Senders (also called publishers) publish messages to there channels which are then independently transmitted to the receivers.

Widely used examples of PubSub systems include WebSub, Kafka and IGMP.

Advantages

Modular

The sender and receiver of messages are decoupled - the publisher doesn't know nor care about which subscribers receive his message and the subscriber doesn't know nor care about which publisher sent the message that he just received. They simply care about channels and are ignorant about the underlying mechanisms for message delivery.

Dynamic

Publishers and subscribers can come and go without affecting the system since they run independently. The messaging system can adapt to changes and keep things running smoothly without interruptions in service.

Scalable

The system has the potential to scale much bigger than traditional client-server messaging systems by taking advantage of better network topology, better routing and message caching among other things.