NETWORK ENCODING

This describes the encoding used to serialize data. It doesn't cover specific objects/messages but focuses on the base types.

The types are not self documenting in any way. They can not be decoded unless you know what they are.

CONVENTIONS

INTEGERS

The integer types used will be named `{signed}{size}{endian}`. For example `u16le` is an unsigned 16 bit integer encoded in little endian byte order while `s64be` is a signed 64 bit integer in big endian. Additionally `u8` and `s8` will represent signed and unsigned bytes respectively. Signed integers use two's complement encoding.

COMPLEX TYPES

This document will use a c-like syntax for describing structures. The structure represents the data that will go over the wire. There will be no padding between the elements and the elements will be sent in the order they appear. For example:

```
struct foo {
        u8    tag;
        u32le data;
}
```

When encoding the values `0x05` and `0x12345678` respectively will appear on the wire as `05 78 56 34 12`.

VARIABLE ARRAYS

Unlike c, length arrays can be used anywhere in structures and will be inline in the protocol. Furthermore the length may be described using an earlier item in the structure.

```
struct blob {
        u32le size;
        u8    data[size];
        u32le checksum;
}
```

This structure is encoded as a 32 bit size, followed by `size` data bytes, then a 32 bit checksum.

PRIMITIVE ALIASES

These types are just aliases for primitive types.

```
// From /src/include/types.h

typedef u32le epoch_t;
typedef u32le ceph_seq_t;
typedef u64le ceph_tid_t;
typedef u64le version_t;
```

STRUCTURES

These are the way structures are encoded. Note that these structures don't actually exist in the source but are the way that different types are encoded.

OPTIONAL

Optionals are represented as a presence byte, followed by the item if it exists.

```
struct ceph_optional<T> {
        u8 present;
        T  element[present? 1 : 0]; // Only if present is non-zero.
}
```

Optionals are used to encode `boost::optional`.

PAIR

Pairs are simply the first item followed by the second.

```
struct ceph_pair<A,B> {
        A a;
        B b;
}
```

Pairs are used to encode `std::pair`.

TRIPLE

Triples are simply the tree elements one after another.

```
struct ceph_triple<A,B,C> {
        A a;
        B b;
        C c;
}
```

Triples are used to encode `ceph::triple`.

LIST

Lists are represented as an element count followed by that many elements.

```
struct ceph_list<T> {
        u32le length;
        T     elements[length];
}
```

Note: The size of the elements in the list are not necessarily uniform.

Lists are used to encode `std::list`, `std::vector`, `std::deque`, `std::set` and `ceph::unordered_set`.

BLOB

A Blob is simply a list of bytes.

```
struct ceph_string {
        ceph_list<u8>;
}

// AKA

struct ceph_string {
        u32le size;
        u8    data[size];
}
```

Blobs are used to encode `std::string`, `const char *` and `bufferlist`.

Note: The content of a Blob is arbratrary binary data.

MAP

Maps are a list of pairs.

```
struct ceph_map<K,V> {
        ceph_list<ceph_pair<K,V>>;
}

// AKA

struct ceph_map<K,V> {
        u32le length;
        ceph_pair<K,V> entries[length];
}
```

Maps are used to encode `std::map`, `std::multimap` and `ceph::unordered_map`.

COMPLEX TYPES

These aren’t hard to find in the source but the common ones are listed here for convenience.

UTIME_T

```
// From /src/include/utime.h
struct utime_t {
        u32le tv_sec;  // Seconds since epoch.
        u32le tv_nsec; // Nanoseconds since the last second.
}
```

CEPH_ENTITY_NAME

```
// From /src/include/msgr.h
struct ceph_entity_name {
        u8    type; // CEPH_ENTITY_TYPE_*
        u64le num;
}

// CEPH_ENTITY_TYPE_* defined in /src/include/msgr.h
```