gafferongames.com

Reliable Ordered Messages

13 min read • original

Hi, I'm Glenn Fiedler and welcome to the fifth article in Building a Game Network Protocol.

It's been quite a while since the last article and in that time I've run ahead and implemented code for the rest this series and created the open source library libyojimbo, a hardened and unit-tested library version of the network protocol described in this article series.

If you want an open-source library to implement reliable messages over UDP for you and much more, check out libyojimbo. But, if you're like me and you want to understand exactly how it all works and maybe implement it yourself, read on because we're going to build up a complete system for sending reliable-ordered messages over UDP from scratch!

Introduction

Many people will tell you that implementing your own reliable message system on top of UDP is foolish. Why write your own poor version of TCP? These people are convinced that any implementation of reliability *inevitably* ends up being a (poor) reimplementation of TCP.

But there are many different ways to implement reliable-messages over UDP, each with different strengths and weaknesses. TCP-lite isn't the only option. In fact, most options for reliable-ordered messages I can

think of work nothing like TCP. So lets get creative and work out how we can take advantage of our situation to implement a reliability system that's *better* than TCP for our purpose.

"Building a Game Network Protocol" is about creating a network protocol for action games like first person shooters, and in my experience, the defining feature for network protocols in this genre is a carrier wave of packets sent in both directions at a steady rate like 20 or 30 packets per-second. These packets contain unreliable-unordered data such as the state of the world at time t, so if a packet with state for time t is lost, resending that packet isn't particularly useful. By the time the resent packet arrives, the time t has already passed.

So this is the situation in which we are implementing reliability. For 90% of our packet data it's best to just drop it and never resend it. For 10% or less (give or take) we actually do need reliability but this data is rare, infrequently sent and much smaller than the unreliable data, on average.

Can we kick TCP's ass in this particular situation? **Absolutely!**

Different Approaches

As mentioned before there are many different options for reliability.

One common approach is to have two packet types: reliable-ordered and unreliable. You'll see this approach in many network libraries. The basic idea is that under the covers the library resends reliable packets until they are received by the other side. This is the option that usually ends up looking like TCP-lite for the reliable-packets. It's not that bad, but you can do much better.

The way I prefer to think of it is that messages are smaller bitpacked elements that know how to serialize themselves. Sent messages are placed in a queue and each time a packet is sent some of the messages

in the send queue are included in the packet. This way there are no reliable packets that need to be resent. Instead, reliable messages are included in unreliable packets until they are received.

The easiest way to do this is to include all unacked messages in each packet sent. It goes something like this: each message sent has an id that increments each time a message is sent. Each outgoing packet includes the start **message id** followed by the data for **n** messages. The receiver continually sends back the most recent received message id to the sender as an ack. Only messages newer than the most recent acked message id are included in packets.

This is simple and easy to implement but if a large burst of packet loss occurs while you are sending messages you get a spike in packet size due to unacked messages. As discussed in packet fragmentation and reassembly sending large packets that require MTU fragmentation results in packet loss amplification. The last thing you want to do while under high packet loss is to increase packet size and induce even more packet loss. It's a potential death spiral.

You can avoid this by extending the system to have an upper bound on the number of messages included in per-packet **n**. But now if you have a high packet send rate (like 60 packets per-second) you are sending the same message multiple times until you get an ack for that message. Maybe you really need this amount of redundancy because your messages are that time critical, but in most cases, your bandwidth would be better spent sending other messages in the send queue (above the maximum messages per-packet **n**) rather than the same set of messages over and over with ultra redundancy.

The approach I prefer combines per-packet acks with a prioritization system that picks the n most important messages to include in each packet. The prioritization function can be anything you wish, but I've found just setting a minimum resend time of 0.1 seconds per-message is enough to enable higher packet send rates without spamming the same

messages over and over. You can get smarter if you wish and make this a function of RTT or do some sort of exponential back-off per-message. The receiver then replies back to the sender with a smart encoding of the set of packets it has received and the sender maintains a data structure that maps a packet sequence number back to the set of messages that were included in that packet, so individual messages are acked when a packet containing them is received by the other side.

Packet Level Acks

Lets move on to implementation!

The foundation of this reliability system is per-packet acks. This just means, the receiver tells the sender which packets it has actually received.

To implement this, add the following header in front of each packet sent:

```
struct Header
{
    uint16_t sequence;
    uint16_t ack;
    uint32_t ack_bits;
};
```

sequence is a number that increases with each packet sent (and wraps around after 65535). **ack** is the most recent packet sequence number received from the other side. **ack_bits** is a bitfield encoding the set of packets received relative to **ack:** if bit **n** is set then packet **ack – n** was received.

Not only is **ack_bits** a smart encoding of acks that saves bandwidth, it adds redundancy to combat packet loss. Each ack is sent 32 times. So if one packet is lost, there's 31 other packets with the same ack. Statistically speaking, acks are very likely to get through.

But bursts of packet loss do happen, so it's important to note that this ack system has the following properties:

1. If you receive an ack for packet n then that packet was definitely received.
2. If you don't receive an ack, the packet was *most likely* not received. But... it might have, and the ack just didn't get through. This is extremely rare.

In my experience it's not necessary to send perfect acks, and building a reliability system on top of a system that very rarely drops acks adds no significant problems. But, if you do implement this system yourself, please, *please* implement a soak test with terrible network conditions to make sure your ack system is working correctly, and by extension, that your message system implementation is actually delivering reliable-ordered messages *reliably and in-order*. You'll find such a soak test in the example source code for this article which is available for patreon supporters, and in the open source network libary libyojimbo.

Sequence Buffers

To implement this ack system we need a data structure on the sender side to track whether a packet has been acked so we can ignore redundant acks (each packet is acked multiple times via **ack_bits**). We also need a data structure on the receiver side to keep track of which packets have been received so we can fill in the **ack_bits** value in the packet header.

The data structure should have the following properties:

- Constant time insertion (inserts may be *random*, for example out of order packets...)
- Constant time query if an entry exists given a packet sequence number
- Constant time access for the data stored for a given packet sequence number
- Constant time removal of entries

You might be thinking. Oh of course, *hash table*. But there's a much simpler way:

```
const int BufferSize = 1024;

uint32_t sequence_buffer[BufferSize];

struct PacketData
{
    bool acked;
};

PacketData packet_data[BufferSize];

PacketData * GetPacketData( uint16_t sequence )
{
    const int index = sequence % BufferSize;
    if ( sequence_buffer[index] == sequence )
        return &packet_data[index];
    else
        return NULL;
}
```

As you can see the trick here is a rolling buffer indexed by sequence number:

```
const int index = sequence % BufferSize;
```

This works because we don't care about being destructive to old entries. As the sequence number increases older entries are naturally overwritten as we insert new ones. The sequence_buffer[index] value is used to test if the entry at that index actually corresponds to the sequence number you're looking for. A sequence buffer value of 0xFFFFFFFF indicates an empty entry and naturally returns NULL for any sequence number query without an extra branch.

When entries are added in order like a send queue, all that needs to be done on insert is to update the sequence buffer value to the new sequence number and overwrite the data at that index:

```
PacketData & InsertPacketData( uint16_t sequence )
{
    const int index = sequence % BufferSize;
    sequence_buffer[index] = sequence;
    return packet_data[index];
}
```

But on the receive side packets arrive out of order and some are lost. Under ridiculously high packet loss (99%) I've seen old sequence buffer entries stick around from before the previous sequence number wrap at 65535 and break my ack logic (leading to false acks and broken reliability where the sender thinks the other side has received something they haven't...)

The solution is to walk between the previous highest insert sequence and the new insert sequence (if it is more recent) and clear those entries in the sequence buffer to 0xFFFFFFFF. Now in the common case, insert is *very close* to constant time, but worst case is linear where n is the number of sequence entries between the previous highest insert sequence and the current insert sequence.

You can do much more with this data structure than just acks. For example, you could extend the per-packet data to include time sent:

```
struct PacketData
{
    bool acked;
    double send_time;
};
```

And with this information you can create your own estimate of round trip time by comparing send time to current time when packets are acked and taking an exponentially smoothed moving average. You can even look at packets in the sent packet sequence buffer older than your RTT estimate (you should have received an ack for them by now...) to create your own packet loss estimate. Handy!

Ack Algorithm

The algorithm for packet level acks is as follows:

On packet send:

1. Insert an entry for for the current send packet sequence number in the sent packet sequence buffer with data indicating that it hasn't been acked yet
2. Generate **ack** and **ack_bits** from the contents of the local received packet sequence buffer and the most recent received packet sequence number
3. Fill the packet header with **sequence, ack** and **ack_bits**
4. Send the packet and increment the send packet sequence number

On packet receive:

1. Read in **sequence** from the packet header
2. If **sequence** is more recent than the previous most recent received packet sequence number, update the most recent received packet sequence number
3. Insert an entry for this packet in the received packet sequence buffer
4. Decode the set of acked packet sequence numbers from **ack** and **ack_bits** in the packet header.
5. Iterate across all acked packet sequence numbers and for any packet that is not already acked call **OnPacketAcked**(uint16_t sequence) and set that packet as 'acked' in the sent packet sequence buffer

Importantly this is done on both sides so if you have a client and a server then the client maintains its own send packet sequence, most recent received sequence and sent packets and received packet sequence buffers and the server maintains the same for each client. It's totally reflective. Each side of the connection runs the same logic, maintaining its own sequence number for sent packets, tracking most recent received packet sequence # from the other side and a sequence buffer of received packets from which it generates **sequence, ack** and **ack_bits** to send to the other side.

And that's really all there is to it. Now you have a callback when a packet is received by the other side: **OnPacketAcked**. The key to this reliability system is that once you know which packets are received, you can build *any* reliability system you want on top. It's not limited just to reliable-ordered messages. For example, you could use it for to know which unreliable state updates actually got through to implement delta encoding on a per-object basis.

Message Objects

Now lets take a look at messages in this system.

Messages are small objects that know how to serialize themselves to and from a bitpacked buffer using a unified serialize function. The serialize function is templated so you write it once and it handles read, write and *measure*.

Yes. Measure. One of my favorite tricks is to have a dummy stream class called **MeasureStream** that doesn't do any actual serialization but just measures the number of bits that *would* be written if you called the serialize function. This is very useful for working out which messages are going to fit into your packet, especially when messages can have arbitrarily complex serialize functions.

```
struct TestMessage : public Message
{
    uint32_t a,b,c;

    TestMessage()
    {
        a = 0;
        b = 0;
        c = 0;
    }

    template <typename Stream> bool Serialize( Stream & stream )
    {
        serialize_bits( stream, a, 32 );
        serialize_bits( stream, b, 32 );
        serialize_bits( stream, c, 32 );
```

```
            return true;
        }

        virtual SerializeInternal( WriteStream & stream )
        {
            return Serialize( stream );
        }

        virtual SerializeInternal( ReadStream & stream )
        {
            return Serialize( stream );
        }

        virtual SerializeInternal( MeasureStream & stream )
        {
            return Serialize( stream );
        }
    };
```

The trick here is to bridge the unified templated serialize function (so you only have to write it once) to virtual serialize methods by calling into it from virtual functions per-stream type. I usually wrap this boilerplate with a macro, but it's expanded in the code above so you can see what's going on.

Now when you have a base message pointer you can do this for any stream type and it just works:

```
Message * message = CreateSomeMessage();
message->SerializeInternal( stream );
```

An alternative if you know the full set of messages at compile time is to implement a big switch statement on message type casting to the correct message type before calling into the serialize function for each type. I've done this in the past on console platform implementations of this message system (eg. PS3 SPUs) but for applications today (2016) I find the overhead of virtual functions to be negligible.

Messages derive from a base class that provide a common interface such as serialization, querying the type of a message and reference counting. Reference counting is necessary because messages are passed around by pointer and stored not only in the message send queue until acked, but also in outgoing packets (which are themselves C++ structs) by pointer.

This is of course all just a strategy to avoid copying data by passing messages and packets around by pointer. Somewhere else (ideally on a separate thread) packets and the messages inside them are serialized to a buffer. Eventually, when no references to a message exist in the message send queue (the message is acked) and no packets including that message remain in the packet send queue, the message is destroyed.

We also need a way to create messages. I do this with a message factory class with a virtual function overriden to create a message by type. It's good if the packet factory knows the total number of message types, so we can serialize a message type over the network with tight bounds and discard malicious packets with message types outside of the valid range.

Here's how I do it:

```
enum TestMessageTypes
{
    TEST_MESSAGE_A,
    TEST_MESSAGE_B,
    TEST_MESSAGE_C,
    TEST_MESSAGE_NUM_TYPES
};

// message definitions omitted

class TestMessageFactory : public MessageFactory
{
public:

    Message * Create( int type )
```

```
    {
        switch ( type )
        {
            case TEST_MESSAGE_A: return new TestMessageA();
            case TEST_MESSAGE_B: return new TestMessageB();
            case TEST_MESSAGE_C: return new TestMessageC();
        }
    }

    virtual int GetNumTypes() const
    {
        return TEST_MESSAGE_NUM_TYPES;
    }
};
```

Again, this is boilerplate and is usually wrapped by macros, but underneath this is what's going on.

Reliable Ordered Message Algorithm

Now lets get down to the details of how to implement reliable-ordered messages on top of the ack system.

The algorithm for sending reliable-ordered messages is as follows:

On message send:

1. Measure how many bits the message serializes to using the measure stream
2. Insert the message pointer and the # of bits it serializes to into a sequence buffer indexed by message id. Set the time that message has last been sent to -1
3. Increment the send message id

On packet send:

1. Walk across the set of messages in the send message sequence buffer between the oldest unacked message id and the most recent inserted message id from left -> right (increasing message id order).
2. **SUPER IMPORTANT:** Never send a message id that the receiver can't buffer or you'll break message acks (since that message won't

be buffered, but the packet containing it will be acked, the sender thinks the message has been received, and will not resend it). This means you must *never* send a message id equal to or more recent than the oldest unacked message id plus the size of the message receive buffer.

3. For any message that hasn't been sent in the last 0.1 seconds and fits in the available space we have left in the packet, add it to the list of messages to send. Messages on the left (older messages) naturally have priority due to the iteration order.
4. Include the messages in the outgoing packet and add a reference to each message. Make sure the packet destructor decrements the ref count for each message.
5. Store the number of messages in the packet **n** and the array of message ids included in the packet in a sequence buffer indexed by the outgoing packet sequence number.
6. Add the packet to the packet send queue.

On packet receive:

1. Walk across the set of messages included in the packet and insert them in the receive message sequence buffer indexed by their message id.
2. The ack system automatically acks the packet sequence number we just received.

On packet ack:

1. Look up the set of messages ids included in the packet by sequence number.
2. Remove those messages from the message send queue if they exist and decrease their ref count.
3. Update the last unacked message id by walking forward from the previous unacked message id in the send message sequence buffer until a valid message entry is found, or you reach the current send message id. Whichever comes first.

On message receive:

1. Check the receive message sequence buffer to see if a message exists for the current receive message id.
2. If the message exists, remove it from the receive message sequence buffer, increment the receive message id and return a pointer to the message.
3. Otherwise, no message is available to receive. Return **NULL**.

In short, messages keep getting included in packets until a packet containing that message is acked. We use a data structure on the sender side to map packet sequence numbers to the set of message ids to ack. Messages are removed from the send queue when they are acked. On the receive side, messages arriving out of order are stored in a sequence buffer indexed by message id, which lets us dequeue them in the order they were sent.

The End Result

This provides the user with an interface that looks something like this on send:

```
TestMessage * message = (TestMessage*) factory.Create( TEST_MESSAGE
if ( message )
{
    message->a = 1;
    message->b = 2;
    message->c = 3;
    connection.SendMessage( message );
}
```

And on the receive side:

```
while ( true )
{
    Message * message = connection.ReceiveMessage();
    if ( !message )
        break;

    if ( message->GetType() == TEST_MESSAGE )
    {
        TestMessage * testMessage = (TestMessage*) message;
```

```
            // process test message
        }

        factory.Release( message );
    }
```

As you can see it couldn't get much simpler.

If this sort of interface is appealing to you, please check out my new open source library libyojimbo.

I hope you're enjoyed the writing in this series so far. Please support my writing on patreon, and I'll write new articles faster, plus you get access to example source code for this article under the BSD open source licence. **Thanks for your support!**

Coming soon: Client Server Connection

In the next article in "Building a Game Network Protocol" show you how to build your own client/server connection layer over UDP that implements challenge/response, assigns clients to slots on the server, rejects client connections when the server is full and detects timeouts.

Original URL:
http://gafferongames.com/building-a-game-network-protocol/reliable-ordered-messages/