

Reliable Ordered Messages

How to implement reliable-ordered messages on top of UDP

Posted by Glenn Fiedler (<http://web.archive.org/web/20181107181424/https://gafferongames.com/about>) on Thursday, September 15, 2016

Introduction

Hi, I'm [Glenn Fiedler](http://web.archive.org/web/20181107181424/https://gafferongames.com/about) (<http://web.archive.org/web/20181107181424/https://gafferongames.com/about>) and welcome to **Building a Game Network Protocol** (<http://web.archive.org/web/20181107181424/https://gafferongames.com/categories/building-a-game-network-protocol/>).

Many people will tell you that implementing your own reliable message system on top of UDP is foolish. After all, why reimplement TCP?

But why limit ourselves to how TCP works? But there are so many different ways to implement reliable-messages and most of them work *nothing* like TCP!

So let's get creative and work out how we can implement a reliable message system that's *better* and *more flexible* than TCP for real-time games.

Different Approaches

A common approach to reliability in games is to have two packet types: reliable-ordered and unreliable. You'll see this approach in many network libraries.

The basic idea is that the library resends reliable packets until they are received by the other side. This is the option that usually ends up looking a bit 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. This makes the most sense when the overhead of length prefixing and padding bitpacked messages up to the next byte is undesirable (eg. lots of small messages included in each packet). 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 outgoing packet. This way there are no reliable packets that need to be resent. Reliable messages are simply included in outgoing packets until they are received.

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 and 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.

You can avoid this by extending the system to have an upper bound on the number of messages included 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.

If your round trip time is 100ms each message will be sent 6 times redundantly before being acked on average. Maybe you really need this amount of redundancy because your messages are extremely time critical, but in most cases, your bandwidth would be better spent on other things.

The approach I prefer combines packet level acks with a prioritization system that picks the n most important messages to include in each packet. This combines time critical delivery and the ability to send only n messages per-packet, while distributing sends across all messages in the send queue.

Packet Level Acks

To implement packet level acks, we add the following packet header:

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

These header elements combine to create the ack system: **sequence** is a number that increases with each packet sent, **ack** is the most recent packet sequence number received, and **ack_bits** is a bitfield encoding the set of acked packets.

If bit n is set in **ack_bits**, then **ack - n** is acked. Not only is **ack_bits** a smart encoding that saves bandwidth, it also adds *redundancy* to combat packet loss. Each ack is sent 32 times. 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:

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 been, and the ack just didn't get through. **This is extremely rare**.

In my experience it's not necessary to send perfect acks. Building a reliability system on top of a system that very rarely drops acks adds no significant problems. But it does create a challenge for testing this system works under all situations because of the edge cases when acks are dropped.

https://gafferongames.com/post/reliable_ordered_messages/ Go

AUG NOV DEC

4 captures

21 Oct 2017 - 7 Nov 2018

07

2017 2018 2019

About this capture

(<http://web.archive.org/web/20181107181424/http://www.patreon.com/gafferongames>) for this article, and the open source network libraries [reliable.io](http://web.archive.org/web/20181107181424/https://github.com/networkprotocol/reliable.io) (<http://web.archive.org/web/20181107181424/https://github.com/networkprotocol/reliable.io>) and [yojimbo](http://web.archive.org/web/20181107181424/http://www.libyojimbo.com/) (<http://web.archive.org/web/20181107181424/http://www.libyojimbo.com/>) which also implement this technique.

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];

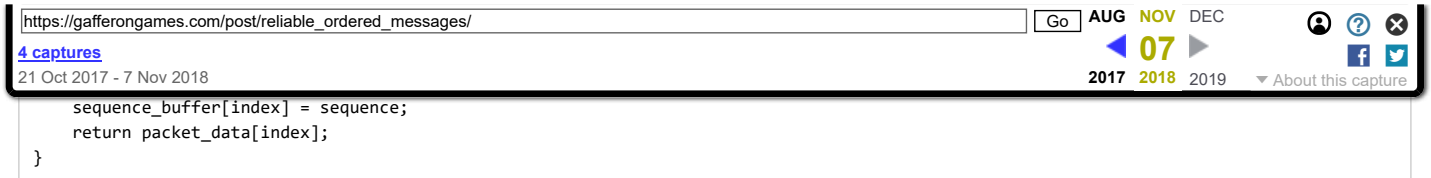
PacketData * GetPacketData( uint16_t sequence )
{
    const int index = sequence % BufferSize;
    if ( sequence_buffer[index] == sequence )
        return &packet_data[index];
    else
        return NULL;
}
```

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:



Unfortunately, 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 to this problem 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.

Before we move on I would like to note that 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;
};
```

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 (http://web.archive.org/web/20181107181424/https://en.wikipedia.org/wiki/Exponential_smoothing). 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.

Ack Algorithm

Now that we have the data structures and packet header, here is the algorithm for implementing packet level acks:

On packet send:

1. Insert an entry 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

https://gafferongames.com/post/reliable_ordered_messages/ Go

AUG NOV DEC

4 captures

21 Oct 2017 - 7 Nov 2018

07

2017 2018 2019

About this capture

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 mark that packet as *acked* in the sent packet sequence buffer.

Importantly this algorithm is done on both sides so if you have a client and a server then 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.

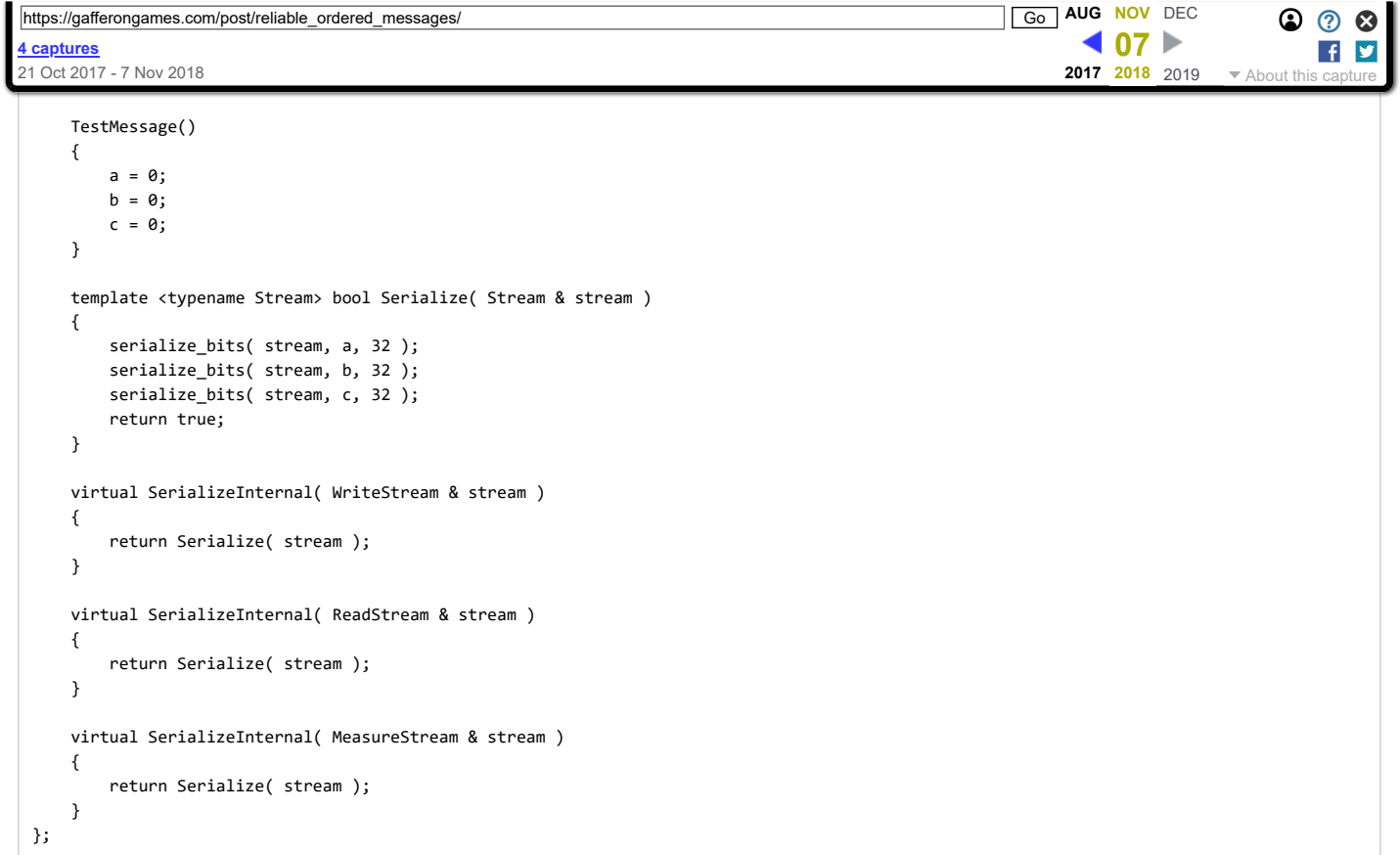
And that's really all there is to it. Now you have a callback when a packet is received by the other side: **OnPacketAcked**. The main benefit of this ack system is now that you know which packets were received, you can build *any* reliability system you want on top. It's not limited to just reliable-ordered messages. For example, you could use it to implement delta encoding on a per-object basis.

Message Objects

Messages are small objects (smaller than packet size, so that many will fit in a typical packet) that know how to serialize themselves. In my system they perform serialization using a [unified serialize function](http://web.archive.org/web/20181107181424/https://gafferongames.com/building-a-game-network-protocol/serialization-strategies) (<http://web.archive.org/web/20181107181424/https://gafferongames.com/building-a-game-network-protocol/serialization-strategies>) 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 particularly useful for working out which messages are going to fit into your packet, especially when messages themselves can have arbitrarily complex serialize functions.



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 and it *just works*:

```

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

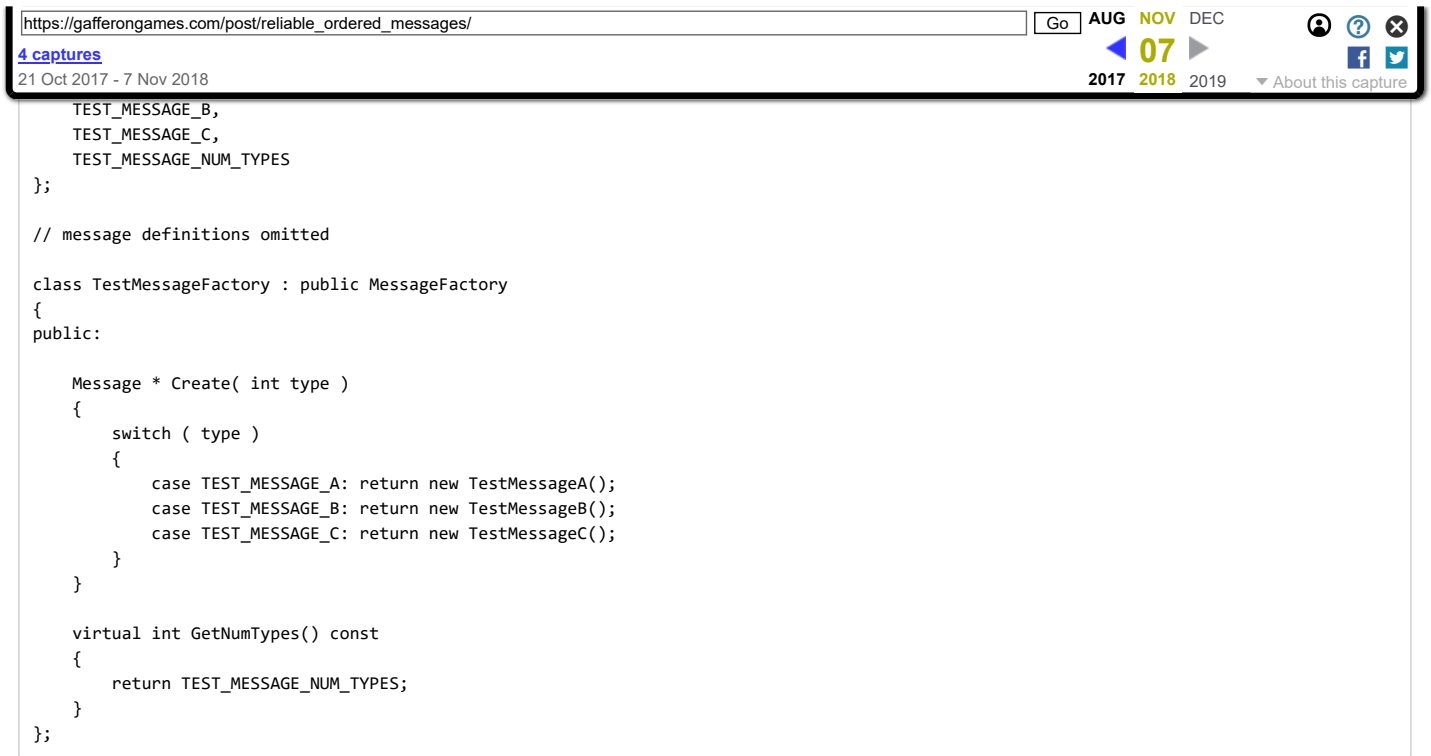
```

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) the overhead of virtual functions is negligible.

Messages derive from a base class that provides 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.

This is a strategy to avoid copying data by passing both 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 overridden to create a message by type. It's good if the packet factory also 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 type values outside of the valid range:



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

Reliable Ordered Message Algorithm

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. 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.

level acks to the set of messages included in that packet.

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 receive 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 );
}
```




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


    if ( message->GetType() == TEST_MESSAGE )
    {
        TestMessage * testMessage = (TestMessage*) message;
        // process test message
    }


    factory.Release( message );
}
```


Which is flexible enough to implement whatever you like on top of it.

NEXT ARTICLE: [Client Server Connection](http://web.archive.org/web/20181107181424/https://gafferongames.com/post/client_server_connection/)

(http://web.archive.org/web/20181107181424/https://gafferongames.com/post/client_server_connection/)

 (<http://web.archive.org/web/20181107181424/https://www.linkedin.com/in/glennfiedler/>)

 (<http://web.archive.org/web/20181107181424/https://twitter.com/gafferongames>)

 (<http://web.archive.org/web/20181107181424/https://github.com/gafferongames>)

Copyright © Glenn Fiedler, 2004 - 2018