Serialization Strategies

Smart tricks that unify packet read and write

Posted by Glenn Fiedler (http://web.archive.org/web/20181107181435/https://gafferongames.com/about) on Sunday, September 4, 2016

Introduction

Hi, I'm Glenn Fiedler (http://web.archive.org/web/20181107181435/https://gafferongames.com/about) and welcome to **Building a Game Network Protocol (http://web.archive.org/web/20181107181435/https://gafferongames.com/categories/building-a-game-network-protocol/)**.

In the previous article (http://web.archive.org/web/20181107181435/https://gafferongames.com/post/reading_and_writing_packets/) we created a bitpacker but it required manual checking to make sure reading a packet from the network is safe. This is a real problem because the stakes are particularly high - a single missed check creates a vulnerability that an attacker can use to crash your server.

In this article, we're going to transform the bitpacker into a system where this checking is *automatic*. We're going to do this with minimal runtime overhead, and in such a way that we don't have to code separate read and write functions, performing both read and write with a single function.

This is called a *serialize function*.

Serializing Bits

Let's start with the goal. Here's where we want to end up:

```
struct PacketA
{
    int x,y,z;

    template <typename Stream> bool Serialize( Stream & stream )
    {
        serialize_bits( stream, x, 32 );
        serialize_bits( stream, y, 32 );
        serialize_bits( stream, z, 32 );
        return true;
    }
};
```

Above you can see a simple serialize function. We serialize three integer variables x,y,z with 32 bits each.

```
    int elements[MaxElements];

    template <typename Stream> bool Serialize( Stream & stream )
    {
        serialize_int( stream, numElements, 0, MaxElements );
        for ( int i = 0; i < numElements; ++i )
        {
            serialize_bits( buffer, elements[i], 32 );
        }
        return true;
    }
};
```

And now something more complicated. We serialize a variable length array, making sure that the array length is in the range [0,MaxElements].

Next, we serialize a rigid body with an simple optimization while it's at rest, serializing only one bit in place of linear and angular velocity:

```
struct RigidBody
{
    vec3f position;
    quat4f orientation;
    vec3f linear_velocity;
    vec3f angular_velocity;

    template <typename Stream> bool Serialize( Stream & stream )
    {
        serialize_vector( stream, position );
        serialize_quaternion( stream, orientation );
        bool at_rest = Stream::IsWriting ? ( velocity.length() == 0 ) : 1;
        serialize_bool( stream, at_rest );
        if ( !at_rest )
        {
            serialize_vector( stream, linear_velocity );
            serialize_vector( stream, angular_velocity );
        }
        else if ( Stream::IsReading )
        {
            linear_velocity = vec3f(0,0,0);
            angular_velocity = vec3f(0,0,0);
        }
        return true;
    }
};
```

Notice how we're able to branch on Stream::IsWriting and Stream::IsReading to write code for each case. These branches are removed by the compiler when the specialized read and write serialize functions are generated.

As you can see, serialize functions are flexible and expressive. They're also *safe*, with each **serialize_*** call performing checks and aborting read if anything is wrong (eg. a value out of range, going past the end of the buffer). Most importantly, this checking is automatic, *so you can't forget to do it!*

Implementation in C++

The trick to making this all work is to create two stream classes that share the same interface: **ReadStream** and **WriteStream**.

The write stream implementation *writes values* using the bitpacker:

```
    enum { IsWriting = 1 };
    enum { IsReading = 0 };

    WriteStream( uint8_t * buffer, int bytes ) : m_writer( buffer, bytes ) {}

    bool SerializeInteger( int32_t value, int32_t min, int32_t max )
    {
        assert( min < max );
        assert( value >= min );
        assert( value <= max );
        const int bits = bits_required( min, max );
        uint32_t unsigned_value = value - min;
        m_writer.WriteBits( unsigned_value, bits );
        return true;
    }

    // ...

private:

    BitWriter m_writer;
};
```

And the read stream implementation *reads values in*:

```
class ReadStream
{
public:

    enum { IsWriting = 0 };
    enum { IsReading = 1 };

    ReadStream( const uint8_t * buffer, int bytes ) : m_reader( buffer, bytes ) {}

    bool SerializeInteger( int32_t & value, int32_t min, int32_t max )
    {
        assert( min < max );
        const int bits = bits_required( min, max );
        if ( m_reader.WouldReadPastEnd( bits ) )
        {
            return false;
        }
        uint32_t unsigned_value = m_reader.ReadBits( bits );
        value = (int32_t) unsigned_value + min;
        return true;
    }

    // ...

private:

    BitReader m_reader;
};
```

With the magic of C++ templates, we leave it up to the compiler to specialize the serialize function to the stream class passed in, producing optimized read and write functions.

To handle safety **serialize_*** calls are not actually functions at all. They're actually macros that return false on error, thus unwinding the stack in case of error, without the need for exceptions.

For example, this macro serializes an integer in a given range:

```
        assert( min < max );                                    \
        int32_t int32_value;                                    \
        if ( Stream::IsWriting )                                \
        {                                                       \
            assert( value >= min );                             \
            assert( value <= max );                             \
            int32_value = (int32_t) value;                      \
        }                                                       \
        if ( !stream.SerializeInteger( int32_value, min, max ) )\
        {                                                       \
            return false;                                       \
        }                                                       \
        if ( Stream::IsReading )                                \
        {                                                       \
            value = int32_value;                                \
            if ( value < min || value > max )                   \
            {                                                   \
                return false;                                   \
            }                                                   \
        }                                                       \
    } while (0)
```

If a value read in from the network is outside the expected range, or we read past the end of the buffer, the packet read is aborted.

Serializing Floating Point Values

We're used to thinking about floating point numbers as being different to integers, but in memory they're just a 32 bit value like any other.

The C++ language lets us work with this fundamental property, allowing us to directly access the bits of a float value as if it were an integer:

```
union FloatInt
{
    float float_value;
    uint32_t int_value;
};

FloatInt tmp;
tmp.float_value = 10.0f;
printf( "float value as an integer: %x\n", tmp.int_value );
```

You may prefer to do this with an aliased uint32_t* pointer, but this breaks with GCC -O2. Friends of mine point out that the only *truly standard way* to get the float as an integer is to cast a pointer to the float value to char* and reconstruct the integer from the bytes values accessed through the char pointer.

Meanwhile in the past 5 years I've had no problems in the field with the union trick. Here's how I use it to serialize an uncompressed float value:

```
{
    union FloatInt
    {
        float float_value;
        uint32_t int_value;
    };
    FloatInt tmp;
    if ( Stream::IsWriting )
    {
        tmp.float_value = value;
    }
    bool result = stream.SerializeBits( tmp.int_value, 32 );
    if ( Stream::IsReading )
    {
        value = tmp.float_value;
    }
    return result;
}
```

This is of course wrapped with a **serialize_float** macro for error checking:

```
#define serialize_float( stream, value )                        \
  do                                                            \
  {                                                             \
      if ( !serialize_float_internal( stream, value ) )         \
      {                                                         \
          return false;                                         \
      }
  } while (0)
```

We can now transmit full precision floating point values over the network.

But what about situations where you don't need full precision? What about a floating point value in the range [0,10] with an acceptable precision of 0.01? Is there a way to send this over the network using less bits?

Yes there is. The trick is to simply divide by 0.01 to get an integer in the range [0,1000] and send that value over the network. On the other side, convert back to a float by multiplying by 0.01.

Here's a general purpose implementation of this basic idea:

```
template <typename Stream>
bool serialize_compressed_float_internal( Stream & stream,
                                          float & value,
                                          float min,
                                          float max,
                                          float res )
{
    const float delta = max - min;
    const float values = delta / res;
    const uint32_t maxIntegerValue = (uint32_t) ceil( values );
    const int bits = bits_required( 0, maxIntegerValue );
    uint32_t integerValue = 0;
    if ( Stream::IsWriting )
    {
        float normalizedValue =
            clamp( ( value - min ) / delta, 0.0f, 1.0f );
        integerValue = (uint32_t) floor( normalizedValue *
                                         maxIntegerValue + 0.5f );
    }
    if ( !stream.SerializeBits( integerValue, bits ) )
    {
        return false;
    }
    if ( Stream::IsReading )
    {
        const float normalizedValue =
            integerValue / float( maxIntegerValue );
        value = normalizedValue * delta + min;
    }
    return true;
}
```

```
#define serialize_compressed_float( stream, value, min, max )       \
  do                                                                \
  {                                                                 \
    if ( !serialize_float_internal( stream, value, min, max ) )     \
    {                                                               \
        return false;                                               \
    }                                                               \
  } while (0)
```

And now the basic interface is complete. We can serialize both compressed and uncompressed floating point values over the network.

Serializing Vectors and Quaternions

Once you can serialize float values it's trivial to serialize vectors over the network. I use a modified version of the vectorial library (http://web.archive.org/web/20181107181435/https://github.com/scoopr/vectorial) in my projects and implement serialization for its vector type like this:

```
template <typename Stream>
bool serialize_vector_internal( Stream & stream,
                                vec3f & vector )
{
    float values[3];
    if ( Stream::IsWriting )
    {
        vector.store( values );
    }
    serialize_float( stream, values[0] );
    serialize_float( stream, values[1] );
    serialize_float( stream, values[2] );
    if ( Stream::IsReading )
    {
        vector.load( values );
    }
    return true;
}

#define serialize_vector( stream, value )                       \
 do                                                             \
 {                                                              \
     if ( !serialize_vector_internal( stream, value ) )         \
     {                                                          \
         return false;                                          \
     }                                                          \
 }                                                              \
 while(0)
```

If your vector is bounded in some range, then you can compress it:

```
template <typename Stream>
bool serialize_compressed_vector_internal( Stream & stream,
                                           vec3f & vector,
                                           float min,
                                           float max,
                                           float res )
{
    float values[3];
    if ( Stream::IsWriting )
    {
        vector.store( values );
    }
    serialize_compressed_float( stream, values[0], min, max, res );
    serialize_compressed_float( stream, values[1], min, max, res );
    serialize_compressed_float( stream, values[2], min, max, res );
    if ( Stream::IsReading )
    {
        vector.load( values );
    }
    return true;
}
```

Serializing Strings and Arrays

What if you need to serialize a string over the network?

Is it a good idea to send a string over the network with null termination? Not really. You're just asking for trouble! Instead, serialize the string as an array of bytes with the string length in front. Therefore, in order to send a string over the network, we have to work out how to send an array of bytes.

First observation. Why waste effort bitpacking an array of bytes into your bit stream just so they are randomly shifted by [0,7] bits? Why not align to byte so you can memcpy the array of bytes directly into the packet?

To align a bitstream just work out your current bit index in the stream and how many bits of padding are needed until the current bit index divides evenly into 8, then insert that number of padding bits.

For bonus points, pad up with zero bits to add entropy so that on read you can verify that yes, you are reading a byte align and yes, it is indeed padded up with zero bits to the next whole byte bit index. If a non-zero bit is discovered in the padding, *abort serialize read and discard the packet*.

Here's my code to align a bit stream to byte:

```
void BitWriter::WriteAlign()
{
    const int remainderBits = m_bitsWritten % 8;
    if ( remainderBits != 0 )
    {
        uint32_t zero = 0;
        WriteBits( zero, 8 - remainderBits );
        assert( ( m_bitsWritten % 8 ) == 0 );
    }
}

bool BitReader::ReadAlign()
{
    const int remainderBits = m_bitsRead % 8;
    if ( remainderBits != 0 )
    {
        uint32_t value = ReadBits( 8 - remainderBits );
        assert( m_bitsRead % 8 == 0 );
        if ( value != 0 )
            return false;
    }
    return true;
}

#define serialize_align( stream )           \
  do                                        \
  {                                         \
      if ( !stream.SerializeAlign() )       \
          return false;                     \
  } while (0)
```

Now we can align to byte prior to writing an array of bytes, letting us use memcpy for the bulk of the array data. The only wrinkle is because the bitpacker works at the word level, it's necessary to have special handling for the head and tail portions. Because of this, the code is quite complex and is omitted for brevity. You can find it in the sample code (http://web.archive.org/web/20181107181435/https://www.patreon.com/gafferongames) for this article.

The end result of all this is a **serialize_bytes** primitive that we can use to serialize a string as a length followed by the string data, like so:

```
                                int buffer_size )
{
    uint32_t length;
    if ( Stream::IsWriting )
    {
        length = strlen( string );
        assert( length < buffer_size - 1 );
    }
    serialize_int( stream, length, 0, buffer_size - 1 );
    serialize_bytes( stream, (uint8_t*)string, length );
    if ( Stream::IsReading )
    {
        string[length] = '\0';
    }
}

#define serialize_string( stream, string, buffer_size )             \
do                                                                  \
{                                                                   \
    if ( !serialize_string_internal( stream,                        \
                                     string,                        \
                                     buffer_size ) )                \
    {                                                               \
        return false;                                               \
    }                                                               \
} while (0)
```

This is an ideal string format because it lets us quickly reject malicious data, vs. having to scan through to the end of the packet searching for **'\0'** before giving up. This is important because otherwise protocol level attacks could be crafted to degrade your server's performance by making it do extra work.

Serializing Array Subsets

When implemeting a game network protocol, sooner or later you need to serialize an array of objects over the network. Perhaps the server needs to send object state down to the client, or there is an array of messages to be sent.

This is straightforward if you are sending *all* objects in the array - just iterate across the array and serialize each object in turn. But what if you want to send a subset of the array?

The simplest approach is to iterate across all objects in the array and serialize a bit per-object if that object is to be sent. If the value of the bit is 1 then the object data follows in the bit stream, otherwise it's ommitted:

```
template <typename Stream>
bool serialize_scene_a( Stream & stream, Scene & scene )
{
    for ( int i = 0; i < MaxObjects; ++i )
    {
        serialize_bool( stream, scene.objects[i].send );
        if ( !scene.objects[i].send )
        {
            if ( Stream::IsReading )
            {
                memset( &scene.objects[i], 0, sizeof( Object ) );
            }
            continue;
        }
        serialize_object( stream, scene.objects[i] );
    }
    return true;
}
```

This approach breaks down as the size of the array gets larger. For example, for an array size of size 4096, then 4096 / 8 = 512 bytes spent on skip bits. That's not good. Can we switch it around so we take overhead propertional to the number of objects sent instead of the total number of objects in the array?

At this point the unified serialize function concept starts to breaks down, and in my opinion, it's best to separate the read and write back into separate functions, because they have so little in common:

```
bool write_scene_b( WriteStream & stream, Scene & scene )
{
    int num_objects_sent = 0;
    for ( int i = 0; i < MaxObjects; ++i )
    {
        if ( scene.objects[i].send )
            num_objects_sent++;
    }
    write_int( stream, num_objects_sent, 0, MaxObjects );
    for ( int i = 0; i < MaxObjects; ++i )
    {
        if ( !scene.objects[i].send )
        {
            continue;
        }
        write_int( stream, i, 0, MaxObjects - 1 );
        write_object( stream, scene.objects[i] );
    }
    return true;
}

bool read_scene_b( ReadStream & stream, Scene & scene )
{
    memset( &scene, 0, sizeof( scene ) );
    int num_objects_sent;
    read_int( stream, num_objects_sent, 0, MaxObjects );
    for ( int i = 0; i < num_objects_sent; ++i )
    {
        int index;
        read_int( stream, index, 0, MaxObjects - 1 );
        read_object( stream, scene.objects[index] );
    }
    return true;
}
```

One more point. The code above walks over the set of objects *twice* on serialize write. Once to determine the number of changed objects and a second time to actually serialize the set of changed objects. Can we do it in one pass instead? Absolutely! You can use another trick, rather than serializing the # of objects in the array up front, use a *sentinel value* to indicate the end of the array:

```
bool write_scene_c( WriteStream & stream, Scene & scene )
{
    for ( int i = 0; i < MaxObjects; ++i )
    {
        if ( !scene.objects[i].send )
        {
            continue;
        }
        write_int( stream, i, 0, MaxObjects );
        write_object( stream, scene.objects[i] );
    }
    write_int( stream, MaxObjects, 0, MaxObjects );
    return true;
}

bool read_scene_c( ReadStream & stream, Scene & scene )
{
    memset( &scene, 0, sizeof( scene ) );
    while ( true )
    {
        int index; read_int( stream, index, 0, MaxObjects );
        if ( index == MaxObjects )
        {
            break;
        }
        read_object( stream, scene.objects[index] );
    }
    return true;
}
```

index costing 12 bits... that's 24000 bits or 3000 bytes (almost 3k!) in your packet wasted on indexing.

You can reduce this overhead by encoding each object index relative to the previous object index. Think about it, you're walking from left to right along an array, so object indices start at 0 and go up to MaxObjects - 1. Statistically speaking, you're quite likely to have objects that are close to each other and if the next index is +1 or even +10 or +30 from the previous one, on average, you'll need quite a few less bits to represent that difference than an absolute index.

Here's one way to encode the object index as an integer relative to the previous object index, while spending less bits on statistically more likely values:

```
                                    int & current )
{
    uint32_t difference;
    if ( Stream::IsWriting )
    {
        assert( previous < current );
        difference = current - previous;
        assert( difference > 0 );
    }

    // +1 (1 bit)
    bool plusOne;
    if ( Stream::IsWriting )
    {
       plusOne = difference == 1;
    }
    serialize_bool( stream, plusOne );
    if ( plusOne )
    {
        if ( Stream::IsReading )
        {
            current = previous + 1;
        }
        previous = current;
        return true;
    }

    // [+2,5] -> [0,3] (2 bits)
    bool twoBits;
    if ( Stream::IsWriting )
    {
        twoBits = difference <= 5;
    }
    serialize_bool( stream, twoBits );
    if ( twoBits )
    {
        serialize_int( stream, difference, 2, 5 );
        if ( Stream::IsReading )
        {
            current = previous + difference;
        }
        previous = current;
        return true;
    }

    // [6,13] -> [0,7] (3 bits)
    bool threeBits;
    if ( Stream::IsWriting )
    {
        threeBits = difference <= 13;
    }
    serialize_bool( stream, threeBits );
    if ( threeBits )
    {
        serialize_int( stream, difference, 6, 13 );
        if ( Stream::IsReading )
        {
            current = previous + difference;
        }
        previous = current;
        return true;
    }

    // [14,29] -> [0,15] (4 bits)
    bool fourBits;
    if ( Stream::IsWriting )
    {
        fourBits = difference <= 29;
    }
    serialize_bool( stream, fourBits );
    if ( fourBits )
    {
        serialize_int( stream, difference, 14, 29 );
        if ( Stream::IsReading )
        {
            current = previous + difference;
        }
        previous = current;
        return true;
    }

    // [30,61] -> [0,31] (5 bits)
    bool fiveBits;
```

```
    serialize_bool( stream, fiveBits );
    if ( fiveBits )
    {
        serialize_int( stream, difference, 30, 61 );
        if ( Stream::IsReading )
        {
            current = previous + difference;
        }
        previous = current;
        return true;
    }

    // [62,125] -> [0,63] (6 bits)
    bool sixBits;
    if ( Stream::IsWriting )
    {
        sixBits = difference <= 125;
    }
    serialize_bool( stream, sixBits );
    if ( sixBits )
    {
        serialize_int( stream, difference, 62, 125 );
        if ( Stream::IsReading )
        {
            current = previous + difference;
        }
        previous = current;
        return true;
    }

    // [126,MaxObjects+1]
    serialize_int( stream, difference, 126, MaxObjects + 1 );
    if ( Stream::IsReading )
    {
        current = previous + difference;
    }
    previous = current;
    return true;
}

template <typename Stream>
bool serialize_scene_d( Stream & stream, Scene & scene )
{
    int previous_index = -1;

    if ( Stream::IsWriting )
    {
        for ( int i = 0; i < MaxObjects; ++i )
        {
            if ( !scene.objects[i].send )
            {
                continue;
            }
            write_object_index( stream, previous_index, i );
            write_object( stream, scene.objects[i] );
        }
        write_object_index( stream, previous_index, MaxObjects );
    }
    else
    {
        while ( true )
        {
            int index;
            read_object_index( stream, previous_index, index );
            if ( index == MaxObjects )
            {
                break;
            }
            read_object( stream, scene.objects[index] );
        }
    }
    return true;
}
```

But what about the worst case? Won't we spent more bits when indices are >= +126 apart than on an absolute index? Yes we do, but how many of these worst case indices fit in an array of size 4096? Just 32. It's nothing to worry about.

We are nearly at the end of this article, and you can see by now that we are sending a completely unattributed binary stream. It's essential that read and write match perfectly, which is of course why the serialize functions are so great, it's hard to desync something when you unify read and write.

But accidents happen, and when they do this system can seem like a stack of cards. What if you somehow desync read and write? How can you debug this? What if somebody tries to connect to your latest server code with an old version of your client?

One technique to protect against this is to include a protocol id in your packet. For example, it could be a combination of a unique number for your game, plus the hash of your protocol version and a hash of your game data. Now if a packet comes in from an incompatible game version, it's automatically discarded because the protocol ids don't match:

```
[protocol id] (64bits)
(packet data)
```

The next level of protection is to pass a CRC32 over your packet and include that in the header. This lets you pick up corrupt packets (these do happen, remember that the IP checksum is just 16 bits...). Now your packet header looks like this:

```
[protocol id] (64bits)
[crc32] (32bits)
(packet data)
```

At this point you may be wincing. Wait. I have to take 8+4 = 12 bytes of overhead per-packet just to implement my own checksum and protocol id? Well actually, *you don't*. You can take a leaf out of how IPv4 does their checksum, and make the protocol id a **magical prefix**.

This means you don't actually send it, and rely on the fact that if the CRC32 is calculated as if the packet were prefixed by the protocol id, then the CRC32 will be incorrect if the sender does not have the same protocol id as the receiver, thus saving 8 bytes per-packet:

```
~~[protocol id] (64bits)~~   // not actually sent, but used to calc crc32
[crc32] (32bits)
(packet data)
```

One final technique, perhaps as much a check against programmer error on your part and malicious senders (although redundant once you encrypt and sign your packet) is the *serialization check*. Basically, somewhere mid-packet, either before or after a complicated serialization section, just write out a known 32 bit integer value, and check that it reads back in on the other side with the same value. If the serialize check value is incorrect *abort read and discard the packet*.

I like to do this between sections of my packet as I write them, so at least I know which part of my packet serialization has desynced read and write as I'm developing my protocol. Another cool trick I like to use is to always serialize a protocol check at the very end of the packet, to detect accidental packet truncation (which happens more often than you would think).

Now the packet looks something like this:

```
[end of packet serialize check] (32 bits)
```

This is great packet structure to use during development.

NEXT ARTICLE: Packet Fragmentation and Reassembly (http://web.archive.org/web/20181107181435/https://gafferongames.com/post/packet_fragmentation_and_reas

---


(http://web.archive.org/web/20181107181435/https://www.linkedin.com/in/glennfiedler/)


(http://web.archive.org/web/20181107181435/https://twitter.com/gafferongames)


(http://web.archive.org/web/20181107181435/https://github.com/gafferongames)