

[gafferongames.com](https://gafferongames.com)

---

# Serialization Strategies

---

21 min read • [original](#)

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

In the [previous article](#) we discussed different ways to read and write packets in multiplayer games. We quickly shot down sending game state via text formats like XML and JSON because they're really inefficient and decided to write own binary protocol instead. We implemented a bitpacker so we don't have to round bools up to 8 bits, solved endianness issues, wrote words at a time instead of bytes and pretty much made the bitpacker as simple and as fast as possible without platform specific tricks.

Where we left off we still had the following problems to solve:

1. We need a way to check if integer values are outside the expected range and abort packet read because people will send malicious packets trying to make us trash memory. The packet read abort must be automatic and not use exceptions because they're really slow.
2. Separate read and write functions are a maintenance nightmare if those functions are coded manually. We'd like to write the serialization code for a packet once but not pay any runtime cost (in terms of additional branching, virtuals and so on) when doing so.

How can we do this? Read on and I'll show you how exactly I do it in C++. It's taken a while for me to develop and refine this technique so I hope you'll find it useful and at least a good alternative to consider vs. the way you currently do it or how you've seen it done in other game engines.

## Unified Packet Serialize Function

Lets 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;
    }
};

struct PacketB
{
    int numElements;
    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;
    }
};

struct PacketC
{
    bool x;
    short y;
    int z;
```

```
template <typename Stream> bool Serialize( Stream & stream )
{
    serialize_int( stream, x, 8 );
    serialize_int( stream, y, 16 );
    serialize_int( stream, z, 32 );
    return true;
}
};
```

Notice there is a single serialize function per-packet struct instead of separate read and write functions. This is great! It halves the amount of serialization code and now you have put in some serious effort in order to desync read and write.

The trick to making this work efficiently is having the stream class templated in the serialize function. There are two stream types in my system: ReadStream and WriteStream. Each class has the same set of methods, but otherwise are not related in any way. One class reads values in from a bit stream to variables, and the other writes variables values out to a bit stream. ReadStream and WriteStream are just wrappers on top of BitReader and BitWriter classes from the previous article.

There are of course alternatives to this approach. If you dislike templates you could have a pure virtual base stream interface and implement that interface with read and write stream classes. But now you're taking a virtual function for each serialize call. Seems like an excessive amount of overhead to me.

Another option is to have an uber-stream class that can be configured to act in read or write mode at runtime. This can be faster than the virtual function method, but you still have to branch per-serialize call to decide if you should read or write so it's not going to be as fast as hand-coded read and write.

I prefer the templated method because it lets the compiler do the work of generating optimized read/write functions for you. You can even code serialize functions like this and let the compiler optimize out a bunch of stuff when specializing read and write:

```
struct Rigidbody
{
    vec3f position;
    quat3f 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 :

        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;
    }
};
```



While this may look inefficient, it's actually not! The template specialization of this function optimizes out all of the branches according to the stream type. Pretty neat huh?

## Bounds Checking and Abort Read

Now that we've twisted the compiler's arm to generate optimized read/write functions, we need some way to automate error checking on read so we're not vulnerable to malicious packets.

The first step is to pass in the range of the integer to the `serialize` function instead of just the number of bits required. Think about it. The `serialize` function can work out the number of bits required from the min/max values:

```
serialize_int( stream, numElements, 0, MaxElements );
```

This opens up the interface to support easy serialization of signed integer quantities and the `serialize` function can check the value read in from the network and make sure it's within the expected range. If the value is outside range, abort `serialize` read immediately and discard the packet.

Since we can't use exceptions to handle this abort (too slow), here's how I like to do it.

In my setup **`serialize_int`** is not actually a function, it's a sneaky macro like this:

```
#define serialize_int( stream, value, min, max )
do
{
    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)
```

The reason I'm being a terrible person here is that I'm using the macro to insert code that checks the result of `SerializeInteger` and returns false on error. This gives you exception-like behavior in the sense that it unwinds the stack back to the top of the serialization callstack on error, but you don't pay anything like the cost of exceptions to do this. The branch to unwind is super uncommon (serialization errors are rare) so branch prediction should have no trouble at all.

Another case where we need to abort is if the stream reads past the end. This is also a rare branch but it's one we do have to check on each serialization operation because reading past the end is undefined. If we fail to do this check, we expose ourselves to infinite loops as we read past the end of the buffer. While it's common to return 0 values when reading past the end of a bit stream (as per-the previous article) there is no guarantee that reading zero values will always result in the `serialize` function terminating correctly if it has loops. This overflow check is necessary for well defined behavior.

One final point. On `serialize write` I don't do any abort on range checks or write past the end of the stream. You can be a lot more relaxed on the write since if anything goes wrong it's pretty much guaranteed to be your fault. Just assert that everything is as expected (in range, not past the end of stream) for each `serialize write` and you're good to go.

## Serializing Floats and Vectors

The bit stream only serializes integer values. How can we serialize a float value?

Seems tricky but it's not actually. A floating point number stored in memory is just a 32 bit value like any other. Your computer doesn't know if a 32 bit word in memory is an integer, a floating point value or

part of a string. It's just a 32 bit value. Luckily, the C++ language (unlike a few others) lets us work with this fundamental property.

You can access the integer value behind a floating point number with a union:

```
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 can also do it via an aliased `uint32_t*` pointer, but I've experienced this break with GCC -O2, so I prefer the union trick instead. Friends of mine point out (likely correctly) that the only *truly standard way* to get the float as an integer is to cast a pointer to the float value to `uint8_t*` and reconstruct the integer value from the four byte values accessed individually through the byte pointer. Seems a pretty dumb way to do it to me though. Ladies and gentlemen... **C++!**

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

```
template <typename Stream>
bool serialize_float_internal( Stream & stream,
                              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;
}

```

Wrap this with a **serialize\_float** macro for convenient error checking on read:

```

#define serialize_float( stream, value )
do
{
    if ( !protocol2::serialize_float_internal( stream, value ) )
        return false;
} while (0)

```

Sometimes you don't want to transmit a full precision float. How can you compress a float value? The first step is to bound that value in some known range then quantize it down to an integer representation.

For example, if you know a floating point number is in range  $[-10, +10]$  and an acceptable resolution for that value is 0.01, then you can just multiply that floating point number by 100.0 to get it in the range  $[-1000, +1000]$  and serialize that as an integer over the network. On the other side, just divide by 100.0 to get back to the floating point value.

Here is a generalized version of this concept:

```

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;
}

```

Once you can serialize float values it's trivial extend to serialize vectors and quaternions over the network. I use a modified version of the awesome [vectorial library](#) for vector math in my projects and I implement serialization for those types 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;
}

```

```
}
```

```
template <typename Stream>
bool serialize_quaternion_internal( Stream & stream,
                                   quat4f & quaternion )
{
    float values[4];
    if ( stream::IsWriting )
        quaternion.store( values );
    serialize_float( stream, values[0] );
    serialize_float( stream, values[1] );
    serialize_float( stream, values[2] );
    serialize_float( stream, values[3] );
    if ( stream::IsReading )
        quaternion.load( values );
    return true;
}
```

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

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

If you know your vector is bounded in some range, you can compress it like this:

```
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;
}
```

If you want to compress an orientation over the network, don't just compress it as a vector with 8.8.8.8 bounded in the range  $[-1, +1]$ . You can do much better if you use the smallest three representation of the quaternion. See the [sample code](#) for this article for an implementation.

## Serializing Strings and Arrays

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

Is it a good idea to send a string over the network with null termination? I don't think so. You're just asking for trouble! Instead, treat the string as an array of bytes with length prefixed. So, in order to send a string over the network, we have to work out how to efficiently 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 shifted by  $[0, 7]$  bits? Why not just align to byte before writing the array, so the array data sits in the packet nicely aligned, each byte of the array corresponding to an actual byte in the packet. You lose only  $[0, 7]$  bits for each array of bytes serialized, depending on the alignment, but that's nothing to be too concerned about in my opinion.

How to align the bit stream to byte? Just work out your current bit index in the stream and how many bits are left to write until the current bit number in the bit stream 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 pad bits, 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 use this align operation to write an array of bytes into the bit stream efficiently: since we are aligned to bytes we can do most of the work using memcpy. The only wrinkle is because the bit reader and bit writer work at the word level, so it's necessary to have special code

to handle the head and tail portion of the byte array, to make sure any previous scratch bits are flushed to memory at the head, and the scratch is properly setup for the next bytes after the array in the tail section.

```
void BitWriter::WriteBytes( const uint8_t* data, int bytes )
{
    assert( GetAlignBits() == 0 );
    assert( m_bitsWritten + bytes * 8 <= m_numBits );
    assert( ( m_bitsWritten % 32 ) == 0 ||
            ( m_bitsWritten % 32 ) == 8 ||
            ( m_bitsWritten % 32 ) == 16 ||
            ( m_bitsWritten % 32 ) == 24 );

    int headBytes = ( 4 - ( m_bitsWritten % 32 ) / 8 ) % 4;
    if ( headBytes > bytes )
        headBytes = bytes;
    for ( int i = 0; i < headBytes; ++i )
        writeBits( data[i], 8 );
    if ( headBytes == bytes )
        return;

    assert( GetAlignBits() == 0 );

    int numWords = ( bytes - headBytes ) / 4;
    if ( numWords > 0 )
    {
        assert( ( m_bitsWritten % 32 ) == 0 );
        memcpy( &m_data[m_wordIndex], data+headBytes, numWords*4 );
        m_bitsWritten += numWords * 32;
        m_wordIndex += numWords;
        m_scratch = 0;
    }

    assert( GetAlignBits() == 0 );

    int tailStart = headBytes + numWords * 4;
    int tailBytes = bytes - tailStart;
    assert( tailBytes >= 0 && tailBytes < 4 );
    for ( int i = 0; i < tailBytes; ++i )
        writeBits( data[tailStart+i], 8 );

    assert( GetAlignBits() == 0 );
```

```
    assert( headBytes + numWords * 4 + tailBytes == bytes );
}

void ReadBytes( uint8_t* data, int bytes )
{
    assert( GetAlignBits() == 0 );
    assert( m_bitsRead + bytes * 8 <= m_numBits );
    assert( ( m_bitsRead % 32 ) == 0 ||
            ( m_bitsRead % 32 ) == 8 ||
            ( m_bitsRead % 32 ) == 16 ||
            ( m_bitsRead % 32 ) == 24 );

    int headBytes = ( 4 - ( m_bitsRead % 32 ) / 8 ) % 4;
    if ( headBytes > bytes )
        headBytes = bytes;
    for ( int i = 0; i < headBytes; ++i )
        data[i] = ReadBits( 8 );
    if ( headBytes == bytes )
        return;

    assert( GetAlignBits() == 0 );

    int numWords = ( bytes - headBytes ) / 4;
    if ( numWords > 0 )
    {
        assert( ( m_bitsRead % 32 ) == 0 );
        memcpy( data + headBytes, &m_data[m_wordIndex], numWords * 4 );
        m_bitsRead += numWords * 32;
        m_wordIndex += numWords;
        m_scratchBits = 0;
    }

    assert( GetAlignBits() == 0 );

    int tailStart = headBytes + numWords * 4;
    int tailBytes = bytes - tailStart;
    assert( tailBytes >= 0 && tailBytes < 4 );
    for ( int i = 0; i < tailBytes; ++i )
        data[tailStart+i] = ReadBits( 8 );

    assert( GetAlignBits() == 0 );

    assert( headBytes + numWords * 4 + tailBytes == bytes );
}
```

```

template <typename Stream>
bool serialize_bytes_internal( Stream & stream,
                               uint8_t* data,
                               int bytes )
{
    return stream.SerializeBytes( data, bytes );
}

#define serialize_bytes( stream, data, bytes ) \
do \
{ \
    if ( !serialize_bytes_internal( stream, data, bytes ) ) \
        return false; \
} while (0)

```

Now we can serialize a string by serializing its length followed by the string data:

```

template <typename Stream>
bool serialize_string_internal( Stream & stream,
                                char* string,
                                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)
```

As you can see, you can build up quite complicated serialization from basic primitives.

## Serializing Array Subsets

When implementing a game network protocol, sooner or later you need to serialize an array of objects over the network. Perhaps the server needs to send all objects down to the client, or an array of events or messages to be sent. This is fairly straightforward if you are sending all objects in the array down to the client, but what if you want to send only a subset of the array?

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

```
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;
}
```



But what if the array of objects is very large, like 4000 objects in the scene?  $4000 / 8 = 500$ . Ruh roh. That's an overhead of 500 bytes, even if you only send one or two objects! That's... not good. Can we switch it around so we take overhead proportional to the number of objects sent instead of the total number of objects in the array?

We can but now, we've done something interesting. We're walking one set of objects in the serialize write (all objects in the array) and are walking over a different set of objects in the serialize read (subset of objects sent). At this point the unified serialize function concept breaks down. It's best to separate the read and write back into separate functions in cases like this:

```
bool write_scene_b( protocol2::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( protocol2::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;
}
```



Alternatively you could generate a separate data structure with the set of changed objects, and implement a serialize for that array of changed objects. But having to generate a C++ data structure for each data structure you want serialized is a huge pain in the ass. Eventually you want to walk several data structures at the same time and effectively write out a dynamic data structure to the bit stream. This is a really common thing to do when writing more advanced serialization methods like delta encoding. As soon as you do it this way, unified serialize no longer makes sense.

My advice is that when you want to do this, don't worry, just separate read and write. Unifying read and write are simply not worth the hassle when dynamically generating a data structure on write. My rule of thumb is that complicated serialization *probably* justifies separate read and write functions, but if possible, try to keep the leaf nodes unified if you can (eg. the actual objects / events, whatever being serialized).

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, a sentinel value to indicate the end of the array, rather than serializing the # of objects in the array up front. This way you can iterate over the array only once on send, and when there are no more objects to send, serialize the sentinel value to indicate the end of the array:

```
bool write_scene_c( protocol2::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( protocol2::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;
}
```

This is pretty simple and it works great if the set of objects sent is a small percentage of total objects. But what if a large number of objects are sent, lets say half of the 4000 objects in the scene. That's 2000 object indices with each index costing 12 bits... that's 24000 bits or 3000 bytes (almost 3k!) in your packet wasted indexing objects.

You can reduce this by encoding each object index relative to the previous object index. Think about it, we're walking left to right along an array, so object indices start at 0 and go up to MaxObjects - 1. Statistically speaking, you're 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 you need to represent 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 (eg. small differences between successive object indices, vs. large ones):

```
template <typename Stream>
bool serialize_object_index_internal( Stream & stream,
                                     int & previous,
                                     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;
if ( Stream::IsWriting )
    fiveBits = difference <= 61;
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;
}
```

In the common case this saves a bunch of bandwidth because object indices tend to be clustered together. In the case where the next object is sent, that's just one bit for the next index being +1 and 5 bits per-index for +2 to +5. On average this gives somewhere between a 2-3X reduction in indexing overhead. But notice that larger indices far apart cost a lot more for each index than the non-relative encoding (12 bits per index). This seems bad but it's not because think about it, even if you hit the 'worst case' (objects indices spaced apart evenly with by +128 apart) how many of these can you actually fit into an object array 4000 large? Just 32. No worries!

## Protocol IDs, CRC32 and Serialization Checks

At this point you may wonder. Wow. This whole thing seems really fragile. It's a totally unattributed binary stream. A stack of cards. What if you somehow desync read and write? What if somebody just sent packets containing random bytes to your server. How long until you hit a sequence of bytes that crashes you out?

I have good news for you and the rest of the game industry since most game servers basically work this way. There are techniques you can use to reduce or virtually eliminate the possibility of corrupt data getting past the serialization layer.

The first technique is to include a protocol id in your packet. Typically, the first 4 bytes you can set to some reasonable rare and unique value, maybe 0x12345678 because nobody else will ever think to use that. But

seriously, put in a hash of your protocol id and your protocol version number in the first 32 bits of each packet and you're doing pretty good. At least if a random packet gets sent to your port from some other application (remember UDP packets can come in from any IP/port combination at any time) you can trivially discard it:

```
[protocol id] (32bits)
(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, and a bunch of stuff will not get picked up by a checksum of 16bits...). Now your packet header looks like this:

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

At this point you may be wincing. Wait. I have to take 8 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*. eg: you don't actually send it, but if both sender and receiver knows the protocol id and the CRC32 is calculated as if the packet were prefixed by the protocol id, the CRC32 will be incorrect if the sender does not have the same protocol id as the receiver, saving 4 bytes per-packet:

```
[protocol id] (32bits) // not actually sent, but used to calc crc3
[crc32] (32bits)
(packet data)
```

Of course CRC32 is only protection against random packet corruption, and is no actual protection against a malicious sender who can easily modify or construct a malicious packet and then properly adjust the CRC32 in the first four bytes. To protect against this you need to use a more cryptographically secure hash function combined with a secret



key perhaps exchanged between client and server over HTTPS by the matchmaker prior to the client attempting to connect to the game server (different key for each client, known only by the server and that particular client).

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 (it's going to happen no matter how hard you try to avoid it...). Another cool trick I like to use is to serialize a protocol check at the very end of the packet, this is super, super useful because it helps pick up packet truncations (like the infamous, little endian vs. big endian truncation of the last word from the [previous article](#)).

So now the packet looks something like this:

```
[protocol id] (32bits)    // not actually sent, but used to calc crc3  
[crc32] (32bits)  
(packet data)  
[end of packet serialize check] (32 bits)
```

You can just compile these protocol checks out in your retail build if you like, especially if you have a good encryption and packet signature, as they should no longer be necessary.

**Up next: [Packet Fragmentation and Reassembly](#)**

Read on for the next article in this series where I show you how to extend your network protocol to perform packet fragmentation and reassembly so you can keep your packet payload under MTU.

[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 BSD open source licence. **Thanks for your support!**



---

**Original URL:**

<http://gafferongames.com/building-a-game-network-protocol/serialization-strategies/>