Reading and Writing Packets

Best practices for reading and writing packets

Posted by Glenn Fiedler (http://web.archive.org/web/20181107181451/https://gafferongames.com/about) on Thursday, September 1, 2016

Introduction

Hi, I'm Glenn Fiedler (http://web.archive.org/web/20181107181451/https://gafferongames.com/about) and welcome to **Building a Game Network Protocol (http://web.archive.org/web/20181107181451/https://gafferongames.com/categories/building-a-game-network-protocol/)**.

In this article we're going to explore how AAA multiplayer games like first person shooters read and write packets. We'll start with text based formats then move into binary hand-coded binary formats and bitpacking.

At the end of this article and the next, you should understand exactly how to implement your own packet read and write the same way the pros do it.

Background

Consider a web server. It listens for requests, does some work asynchronously and sends responses back to clients. It's stateless and generally not real-time, although a fast response time is great. Web servers are most often IO bound.

Game server are different. They're a headless version of the game running in the cloud. As such they are stateful and CPU bound. The traffic patterns are different too. Instead of infrequent request/response from tens of thousands of clients, a game server has far fewer clients, but processes a continuous stream of input packets sent from each client 60 times per-second, and broadcasts out the state of the world to clients 10, 20 or even 60 times per-second.

And this state is **huge**. Thousands of objects with hundreds of properties each. Game network programmers spend a lot of their time optimizing exactly how this state is sent over the network with crazy bit-packing tricks, hand-coded binary formats and delta encoding.

What would happen if we just encoded this world state as XML?

```
    <property name="orientation" value="(1,0,0,0)"</property>
    <property name="velocity" value="(10,0,0)"</property>
    <property name="health" value="100"></property>
    <property name="weapon" value="110"></property>
    ... 100s more properties per-object ...
 </object>
 <object id="100" class="grunt">
   <property name="position" value="(100,100,0)"</property>
   <property name="health" value="10"</property>
 </object>
 <object id="110" class="weapon">
   <property type="semi-automatic"></property>
   <property ammo_in_clip="8"></property>
   <property round_in_chamber="true"></property>
 </object>
 ... 1000s more objects ...
</world_update>
```

Pretty verbose... it's hard to see how this would be practical for a large world.

JSON is a bit more compact:

```
{
  "world_time": 0.0,
  "objects": {
    1: {
      "class": "player",
      "position": "(0,0,0)",
      "orientation": "(1,0,0,0)",
      "velocity": "(10,0,0)",
      "health": 100,
      "weapon": 110
    }
    100: {
      "class": "grunt",
      "position": "(100,100,0)",
      "health": 10
    }
    110: {
      "class": "weapon",
      "type: "semi-automatic"
      "ammo_in_clip": 8,
      "round_in_chamber": 1
    }
    // etc...
  }
}
```

But it still suffers from the same problem: the description of the data is larger than the data itself. What if instead of fully describing the world state in each packet, we split it up into two parts?

1. A schema that describes the set of object classes and properties per-class, **sent only once** when a client connects to the server.

2. Data sent rapidly from server to client, **which is encoded relative to the schema**.

The schema could look something like this:

```
      "properties": {
        0: {
          "name": "position",
          "type": "vec3f"
        }
        1: {
          "name": "orientation",
          "type": "quat4f"
        }
        2: {
          "name": "velocity",
          "type": "vec3f"
        }
        3: {
          "name": "health",
          "type": "float"
        }
        4: {
          "name": "weapon",
          "type": "object",
        }
      }
    }
    1: "grunt": {
      "properties": {
        0: {
          "name": "position",
          "type": "vec3f"
        }
        1: {
          "name": "health",
          "type": "float"
        }
      }
    }
    2: "weapon": {
      "properties": {
        0: {
          "name": "type",
          "type": "enum",
          "enum_values": [ "revolver", "semi-automatic" ]
        }
        1: {
          "name": "ammo_in_clip",
          "type": "integer",
          "range": "0..9",
        }
        2: {
          "name": "round_in_chamber",
          "type": "integer",
          "range": "0..1"
        }
      }
    }
  }
}
```

The schema is quite big, but that's beside the point. It's sent only once, and now the client knows the set of classes in the game world and the number, name, type and range of properties per-class.

With this knowledge we can make the rapidly sent portion of the world state much more compact:

```
{
  "world_time": 0.0,
  "objects": {
    1: [0,"(0,0,0)","(1,0,0,0)","(10,0,0)",100,110],
    100: [1,"(100,100,0)",10],
    110: [2,1,8,1]
  }
}
```

```
0.0
1:0,0,0,0,1,0,0,0,10,0,0,100,110
100:1,100,100,0,10
110:2,1,8,1
```

As you can see, it's much more about what you **don't send** than what you do.

The Inefficiencies of Text

We've made good progress on our text format so far, moving from a highly attributed stream that fully describes the data (more description than actual data) to an unattributed text format that's an order of magnitude more efficient.

But there are inherent inefficiencies when using text format for packets:

- We are most often sending data in the range **A-Z**, **a-z** and **0-1**, plus a few other symbols. This wastes the remainder of the **0-255** range for each character sent. From an information theory standpoint, this is an inefficient encoding.

- The text representation of integer values are in the general case much less efficient than the binary format. For example, in text format the worst case unsigned 32 bit integer **4294967295** takes 10 bytes, but in binary format it takes just four.

- In text, even the smallest numbers in **0-9** range require at least one byte, but in binary, smaller values like **0, 11, 31, 100** can be sent with fewer than 8 bits if we know their range ahead of time.

- If an integer value is negative, you have to spend a whole byte on **'-'** to indicate that.

- Floating point numbers waste one byte specifying the decimal point.

- The text representation of numerical values are variable length: **"5"**, **"12345"**, **"3.141593".** Because of this we need to spend one byte on a separator after each value so we know when it ends.

- Newlines **'\n'** or some other separator are required to distinguish between the set of variables belonging to one object and the next. When you have thousands of objects, this really adds up.

In short, if we wish to optimize any further, it's necessary to switch to a binary format.

Switching to a Binary Format

In the web world there are some really great libraries that read and write binary formats like BJSON (http://web.archive.org/web/20181107181451/http://bjson.org/), Protocol Buffers (http://web.archive.org/web/20181107181451/https://developers.google.com/protocol-buffers/), Flatbuffers (http://web.archive.org/web/20181107181451/https://google.github.io/flatbuffers/), Thrift (http://web.archive.org/web/20181107181451/https://thrift.apache.org/), Cap'n Proto (http://web.archive.org/web/20181107181451/https://capnproto.org/) and MsgPack (http://web.archive.org/web/20181107181451/http://msgpack.org/index.html).

the gold standard.

There are a few reasons for this. Web binary formats are designed for situations where versioning of data is *extremely* important. If you upgrade your backend, older clients should be able to keep talking to it with the old format. Data formats are also expected to be language agnostic. A backend written in Golang should be able to talk with a web client written in JavaScript and other server-side components written in Python or Java.

Game servers are completely different beasts. The client and server are almost always written in the same language (C++), and versioning is much simpler. If a client with an incompatible version tries to connect, that connection is simply rejected. There's simply no need for compatibility across different versions.

So if you don't need versioning and you don't need cross-language support what are the benefits for these libraries? Convenience. Ease of use. Not needing to worry about creating, testing and debugging your own binary format.

But this convenience is offset by the fact that these libraries are less efficient and less flexible than a binary protocol we can roll ourselves. So while I encourage you to evaluate these libraries and see if they suit your needs, for the rest of this article, we're going to move forward with a custom binary protocol.

Getting Started with a Binary Format

One option for creating a custom binary protocol is to use the in-memory format of your data structures in C/C++ as the over-the-wire format. People often start here, so although I don't recommend this approach, lets explore it for a while before we poke holes in it.

First define the set of packets, typically as a union of structs:

```
struct Packet
{
    enum PacketTypeEnum { PACKET_A, PACKET_B, PACKET_C };

    uint8_t packetType;

    union
    {
        struct PacketA
        {
            int x,y,z;
        } a;

        struct PacketB
        {
            int numElements;
            int elements[MaxElements];
        } b;

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

reverse: read in the first byte, then according to the packet type, copy the packet data to the corresponding struct.

It couldn't get simpler. So why do most games avoid this approach?

The first reason is that different compilers and platforms provide different packing of structs. If you go this route you'll spend a lot of time with **#pragma pack** trying to make sure that different compilers and different platforms lay out the structures in memory exactly the same way.

The next one is endianness. Most computers are mostly little endian (http://web.archive.org/web/20181107181451/https://en.wikipedia.org/wiki/Endianness#Little-endian) these days but historically some architectures like PowerPC were big endian (http://web.archive.org/web/20181107181451/https://en.wikipedia.org/wiki/Endianness#Big-endian). If you need to support communication between little endian and big endian machines, the memcpy the struct in and out of the packet approach simply won't work. At minimum you need to write a function to swap bytes between host and network byte order on read and write for each variable in your struct.

There are other issues as well. If a struct contains pointers you can't just serialize that value over the network and expect a valid pointer on the other side. Also, if you have variable sized structures, such as an array of 32 elements, but most of the time it's empty or only has a few elements, it's wasteful to always send the array at worst case size. A better approach would support a variable length encoding that only sends the actual number of elements in the array.

But ultimately, what really drives a stake into the heart of this approach is **security**. It's a *massive* security risk to take data coming in over the network and trust it, and that's exactly what you do if you just copy a block of memory sent over the network into your struct. Wheee! What if somebody constructs a malicious **PacketB** and sends it to you with **numElements** = 0xFFFFFFFF?

You should, no you *must*, at minimum do some sort of per-field checking that values are in range vs. blindly accepting what is sent to you. This is why the memcpy struct approach is rarely used in professional games.

Read and Write Functions

The next level of sophistication is read and write functions per-packet.

Start with the following simple operations:

```
void WriteInteger( Buffer & buffer, uint32_t value );
void WriteShort( Buffer & buffer, uint16_t value );
void WriteChar( Buffer & buffer, uint8_t value );

uint32_t ReadInteger( Buffer & buffer );
uint16_t ReadShort( Buffer & buffer );
uint8_t ReadByte( Buffer & buffer );
```

These operate on a structure which keeps track of the current position:

```
    int size;           // size of buffer data (bytes)
    int index;          // index of next byte to be read/written
};
```

The write integer function looks something like this:

```
void WriteInteger( Buffer & buffer, uint32_t value )
{
    assert( buffer.index + 4 <= size );
#ifdef BIG_ENDIAN
    *((uint32_t*)(buffer.data+buffer.index)) = bswap( value );
#else // #ifdef BIG_ENDIAN
    *((uint32_t*)(buffer.data+buffer.index)) = value;
#endif // #ifdef BIG_ENDIAN
    buffer.index += 4;
}
```

And the read integer function looks like this:

```
uint32_t ReadInteger( Buffer & buffer )
{
    assert( buffer.index + 4 <= size );
    uint32_t value;
#ifdef BIG_ENDIAN
    value = bswap( *((uint32_t*)(buffer.data+buffer.index)) );
#else // #ifdef BIG_ENDIAN
    value = *((uint32_t*)(buffer.data+buffer.index));
#endif // #ifdef BIG_ENDIAN
    buffer.index += 4;
    return value;
}
```

Now, instead of copying across packet data in and out of structs, we implement read and write functions for each packet type:

```
    void Write( Buffer & buffer )
    {
        WriteInteger( buffer, x );
        WriteInteger( buffer, y );
        WriteInteger( buffer, z );
    }

    void Read( Buffer & buffer )
    {
        ReadInteger( buffer, x );
        ReadInteger( buffer, y );
        ReadInteger( buffer, z );
    }
};

struct PacketB
{
    int numElements;
    int elements[MaxElements];

    void Write( Buffer & buffer )
    {
        WriteInteger( buffer, numElements );
        for ( int i = 0; i < numElements; ++i )
            WriteInteger( buffer, elements[i] );
    }

    void Read( Buffer & buffer )
    {
        ReadInteger( buffer, numElements );
        for ( int i = 0; i < numElements; ++i )
            ReadInteger( buffer, elements[i] );
    }
};

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

    void Write( Buffer & buffer )
    {
        WriteByte( buffer, x );
        WriteShort( buffer, y );
        WriteInt( buffer, z );
    }

    void Read( Buffer & buffer )
    {
        ReadByte( buffer, x );
        ReadShort( buffer, y );
        ReadInt( buffer, z );
    }
};
```

When reading and writing packets, start the packet with a byte specifying the packet type via ReadByte/WriteByte, then according to the packet type, call the read/write on the corresponding packet struct in the union.

Now we have a system that allows machines with different endianness to communicate and supports variable length encoding of elements.

Bitpacking

boolean values? It would be nice to send these as one bit instead of 8!

One way to implement this is to manually organize your C++ structures into packed integers with bitfields and union tricks, such as grouping all bools together into one integer type via bitfield and serializing them as a group. But this is tedious and error prone and there's no guarantee that different C++ compilers pack bitfields in memory exactly the same way.

A much more flexible way that trades a small amount of CPU on packet read and write for convenience is a **bitpacker**. This is code that reads and writes non-multiples of 8 bits to a buffer.

Writing Bits

Many people write bitpackers that work at the byte level. This means they flush bytes to memory as they are filled. This is simpler to code, but the ideal is to read and write words at a time, because modern machines are optimized to work this way instead of farting across a buffer at byte level like it's 1985.

If you want to write 32 bits at a time, you'll need a scratch word twice that size, eg. uint64_t. The reason is that you need the top half for overflow. For example, if you have just written a value 30 bits long into the scratch buffer, then write another value that is 10 bits long you need somewhere to store 30 + 10 = 40 bits.

```
uint64_t scratch;
int scratch_bits;
int word_index;
uint32_t * buffer;
```

When we start writing with the bitpacker, all these variables are cleared to zero except **buffer** which points to the start of the packet we are writing to. Because we're accessing this packet data at a word level, not byte level, make sure packet buffers lengths are a multiple of 4 bytes.

Let's say we want to write 3 bits followed by 10 bits, then 24. Our goal is to pack this tightly in the scratch buffer and flush that out to memory, 32 bits at a time. Note that 3 + 10 + 24 = 37. We have to handle this case where the total number of bits don't evenly divide into 32. This is actually the *common case*.

At the first step, write the 3 bits to **scratch** like this:

```
xxx
```

scratch_bits is now 3.

Next, write 10 bits:

```
yyyyyyyyyyxxx
```

scratch_bits is now 13 (3+10).

Next write 24 bits:

scratch_bits is now 37 (3+10+24). We're straddling the 32 bit word boundary in our 64 bit **scratch** variable and have 5 bits in the upper 32 bits (overflow). Flush the lower 32 bits of **scratch** to memory, advance **word_index** by one, shift **scratch** right by 32 and subtract 32 from **scratch_bits**.

scratch now looks like this:

```
zzzzz
```

We've finished writing bits but we still have data in scratch that's not flushed to memory. For this data to be included in the packet we need to make sure to flush any remaining bits in **scratch** to memory at the end of writing.

When we flush a word to memory it is converted to little endian byte order. To see why this is important consider what happens if we flush bytes to memory in big endian order:

```
DCBA000E
```

Since we fill bits in the word from right to left, the last byte in the packet E is actually on the right. If we try to send this buffer in a packet of 5 bytes (the actual amount of data we have to send) the packet catches 0 for the last byte instead of E. Ouch!

But when we write to memory in little endian order, bytes are reversed back out in memory like this:

```
ABCDE000
```

And we can write 5 bytes to the network and catch E at the end. Et voilà!

Reading Bits

To read the bitpacked data, start with the buffer sent over the network:

```
ABCDE
```

The bit reader has the following state:

```
uint64_t scratch;
int scratch_bits;
int total_bits;
int num_bits_read;
int word_index;
uint32_t * buffer;
```

To start all variables are cleared to zero except **total_bits** which is set to the size of the packet as bytes * 8, and **buffer** which points to the start of the packet.

The value of **scratch** is now:

```
zzzzzzzzzzzzzzzzzzzyyyyyyyyyyxxx
```

Read off the low 3 bits, giving the expected value of:

```
xxx
```

Shift **scratch** to the right 3 bits and subtract 3 from **scratch_bits**:

```
zzzzzzzzzzzzzzzzzzzyyyyyyyyyy
```

Read off another 10 bits in the same way, giving the expected value of:

```
yyyyyyyyyy
```

Scratch now looks like:

```
zzzzzzzzzzzzzzzzzz
```

The next read asks for 24 bits but **scratch_bits** is only 19 (=32-10-3).

It's time to read in the next word. Shifting the word in memory left by **scratch_bits** (19) and or it on top of **scratch**.

Now we have all the bits necessary for z in **scratch**:

```
zzzzzzzzzzzzzzzzzzzzzzz
```

Read off 24 bits and shift **scratch** right by 24. **scratch** is now all zeros.

We're done!

Beyond Bitpacking

Reading and writing integer values into a packet by specifying the number of bits to read/write is not the most user friendly option.

Consider this example:

```
{
    int numElements;
    int elements[MaxElements];

    void Write( BitWriter & writer )
    {
        WriteBits( writer, numElements, 6 );
        for ( int i = 0; i < numElements; ++i )
            WriteBits( writer, elements[i] );
    }

    void Read( BitReader & reader )
    {
        ReadBits( reader, numElements, 6 );
        for ( int i = 0; i < numElements; ++i )
            ReadBits( reader, elements[i] );
    }
};
```

This code looks fine at first glance, but let's assume that some time later you, or somebody else on your team, increases **MaxElements** from 32 to 200 but forget to update the number of bits required to **7**. Now the high bit of **numElements** are being silently truncated on send. It's pretty hard to track something like this down after the fact.

The simplest option is to just turn it around and define the maximum number of elements in terms of the number of bits sent:

```
const int MaxElementBits = 7;
const int MaxElements = ( 1 << MaxElementBits ) - 1;
```

Another option is to get fancy and work out the number of bits required at compile time:

```
template <uint32_t x> struct PopCount
{
    enum { a = x - ( ( x >> 1 ) & 0x55555555 ),
           b = ( ( ( a >> 2 ) & 0x33333333 ) + ( a & 0x33333333 ) ),
           c = ( ( ( b >> 4 ) + b ) & 0x0f0f0f0f ),
           d = c + ( c >> 8 ),
           e = d + ( d >> 16 ),
    result = e & 0x0000003f };
};

template <uint32_t x> struct Log2
{
    enum { a = x | ( x >> 1 ),
           b = a | ( a >> 2 ),
           c = b | ( b >> 4 ),
           d = c | ( c >> 8 ),
           e = d | ( d >> 16 ),
           f = e >> 1,
    result = PopCount<f>::result };
};

template <int64_t min, int64_t max> struct BitsRequired
{
    static const uint32_t result =
        ( min == max ) ? 0 : ( Log2<uint32_t(max-min)>::result + 1 );
};

#define BITS_REQUIRED( min, max ) BitsRequired<min,max>::result
```

Now you can't mess up the number of bits, and you can specify non-power of two maximum values and it everything works out.

But be careful when array sizes aren't a power of two! In the example above **MaxElements** is 32, so **MaxElementBits** is 6. This seems fine because all values in [0,32] fit in 6 bits. The problem is that there are additional values within 6 bits that are *outside* our array bounds: [33,63]. An attacker can use this to construct a malicious packet that corrupts memory!

This leads to the *inescapable* conclusion that it's not enough to just specify the number of bits required when reading and writing a value, we must also check that it is within the valid range: [min,max]. This way if a value is outside of the expected range we can detect that and abort read.

I used to implement this using C++ exceptions, but when I profiled, I found it to be incredibly slow. In my experience, it's much faster to take one of two approaches: set a flag on the bit reader that it should abort, or return false from read functions on failure. But now, in order to be completely safe on read you must to check for error on every read operation.

```
const int MaxElements = 32;
const int MaxElementBits = BITS_REQUIRED( 0, MaxElements );

struct PacketB
{
    int numElements;
    int elements[MaxElements];

    void Write( BitWriter & writer )
    {
        WriteBits( writer, numElements, MaxElementBits );
        for ( int i = 0; i < numElements; ++i )
        {
            WriteBits( writer, elements[i], 32 );
        }
    }

    void Read( BitReader & reader )
    {
        ReadBits( reader, numElements, MaxElementBits );

        if ( numElements > MaxElements )
        {
            reader.Abort();
            return;
        }

        for ( int i = 0; i < numElements; ++i )
        {
            if ( reader.IsOverflow() )
                break;

            ReadBits( buffer, elements[i], 32 );
        }
    }
};
```

If you miss any of these checks, you expose yourself to buffer overflows and infinite loops when reading packets. Clearly you don't want this to be a manual step when writing a packet read function. *You want it to be automatic.*

NEXT ARTICLE: Serialization Strategies (http://web.archive.org/web/20181107181451/https://gafferongames.com/post/serialization_strategies/)

(http://web.archive.org/web/20181107181451/https://www.linkedin.com/in/glennfiedler/)

(http://web.archive.org/web/20181107181451/https://twitter.com/gafferongames)

(http://web.archive.org/web/20181107181451/https://github.com/gafferongames)