gafferongames.com

Reading and Writing Packets

17 min read • original

Introduction

Hi, I'm Glenn Fiedler. I'm a professional game network programmer with over 15 years experience working in the game industry. These days I run The Network Protocol Company where I help people network their games, and in my spare time write open source code and articles like the one you are reading right now. Before I started my own company, I worked as a software engineer at **Irrational Games**, **Sony Santa Monica**, **Respawn Entertainment** and many others.

Welcome to **Building a Game Network Protocol**. In this article series I'm building up a professional-grade client/server network protocol for action games like first person shooters. The best practice is to network these games with a custom network protocol built on top of UDP. I'm going to show you why this is, what this protocol looks like, and how you can build one yourself!

If you come from a web development background you've probably used XML, JSON or YAML to send data over the network in text format. This works reasonably well in the web development world, but in game networking, text protocols are rarely used. In this article I'm going to show you why.

Consider a web server. It listens for requests, does work asynchronously and sends responses back to clients. It's stateless and generally not real-time (although fast response time is great). In contrast, a game server is runs a real-time simulation of a game world and is completely stateful. These are totally different situations!

The network traffic patterns are different too. Instead of infrequent requests from tens of thousands of clients, a game server has fewer clients, but must process a continuous stream of input packets sent from each client 60 times per-second. It takes this input and steps the simulation forward, then at some rate (20, 30 or even 60 times per-second) broadcasts the state of the world to each client.

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

But before we get to all this, what would happen if we encoded this world state as XML?

```
<world_update world_time="0.0">
  <object id="1" class="player">
    <property name="position" value="(0,0,0)"</property>
    <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>
```

As you can see, we spend more of the XML file *describing* the data than on actual data!

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
    }
  }
}
```

But it's still not great...

What if, instead of fully describing the world state in each packet, we split it 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 that is sent rapidly (20, 30 or 60 times per-second) from server to client, which is encoded relative to the schema.

A schema could look something like this:

```
{
  "classes": {
    0: "player" {
      "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"
          }
        }
      }
    }
}
```

Now the client knows the set of classes in the game world and the number, name, type and range of properties for each class. For example, if an object is an instance of class 2 then we know it's a weapon, and property 0 on that object is the weapon type: revolver (0) or semi-automatic (1).

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]
  }
}
```

And we can compress even further by switching to a custom text format:

```
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, with game network programming it's more about what you don't send than what you do. The trick here is when the client and server both know something about the structure of the data, large portions can be omitted that would otherwise need to be sent. This is one case where being stateful is a good thing.

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 packet data:

1. 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.
2. 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.
3. In text, even the smallest numbers in **0-9** range require at least one byte. 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.
4. If an integer value is negative, you have to spend a whole byte on **'-'** to indicate that.
5. Floating point numbers waste one byte specifying the decimal point **'.'**
6. 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. No separators are required in binary because the number of bits required can be determined as a function of the range of its possible values, which is known by both sides.

7. 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 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, Protocol Buffers, Flatbuffers, Thrift, Cap'n Proto and MsgPack. In some cases, these libraries can be a great fit for building your game network protocol.

But in others, especially in first person shooters where efficiency is paramount, a hand-tuned binary protocol designed specifically for the task at hand is a much better option.

There are a few reasons for this. Web binary formats are designed for a situation 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). Language agnostic data formats are also key; a backend written in golang should be able to talk with a web client running inside a browser in JavaScript and microservices written in different languages should be able to communicate without having to upgrade all components at the same time.

Game servers for AAA FPS games are completely different beasts. The client and server are almost always written in the same language (C++) because large portions of the game code must run identically on the server and the client for client-side prediction. Versioning is much simpler as well, because if a client with an incompatible game version tries to connect to a game server, that connection is simply rejected. This may seem like a problem, but in practice, game clients are typically required to update to new client versions before they can play online anyway and in development your matchmaker can ensure that clients are sent to game server instances with compatible game 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 than a binary protocol we can roll ourselves. We know more about the data we are sending than any library writer ever could and we can take advantage of this to send less bits by developing our own custom binary format.

Getting Started with a Binary Format

One option for creating your own 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;
    };
};
```

When writing the packet, set the first byte in the packet to the packet type number (0,1 or 2). Then depending on the packet type, memcpy the appropriate union struct into the packet. On read do the 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. Not that it's impossible, but it's certainly not trivial.

The next one is endianness. Most computers are mostly little endian these days (Intel) but PowerPC cores are big endian. Historically, network data was sent over the wire in big endian format (network byte order) but there is no reason for you to follow this tradition. Modern game network protocols write in little endian order to minimize the amount of work in the common case.

But if you do 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.

Ultimately, the issue that 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 I can't think of anything more trusting than taking a block of data sent to you over the network and just copying it into a struct. Wheee! What if somebody constructs a malicious packet and sends it to you with 0xFFFFFFFF in the **numElements** of **PacketB** causing you to trash all over memory when you process that packet?

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.

Upgrading to 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 buffer structure which keeps track of the current position in the packet:

```
struct Buffer
{
    uint8_t * data;     // pointer to buffer data
```

```
    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, implement read and write functions for each packet type:

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

    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.

This is real progress. Now we have a system that allows machines with different endianness to communicate *and* supports variable length encoding of elements within structures.

Bitpacking

So far we have been reading and writing packets at the byte level.

But if we have a value in the range [0,1000] we really only need 10 bits to represent all possible values. Wouldn't it be nice if we could write just 10 bits, instead of rounding up to 16?

It would be really nice if we could send boolean values using only one bit, instead of a byte.

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 union 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 some small amount of CPU on packet read and write for convenience is a **bitpacker**. This is code that is able to read and write non-multiples of 8 bits into a buffer. To do this,

it must perform a bunch of manual bit-shifting because data being written and read doesn't correspond to any natural machine word type (byte, short or int).

Writing Bits

Many (most?) 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, at least until you flush the top 32 bits out to memory.

We're going to work at the 32 bit word level, so the bitpacker state for write looks something like this:

```
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:

```
zzzzzzzzzzzzzzzzzzzzzzzzyyyyyyyyyyxxx
```

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 yet. It's important to remember to manually flush any remaining bits in **scratch** to memory at the end of write.

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 in bytes * 8, and **buffer** which points to the start of the packet.

The user requests a read of 3 bits. Since **scratch_bits** is zero, it's time to read in the first word. Take that word and stash it in **scratch**, shifted left by **scratch_bits** (which is zero...). Add 32 to **scratch_bits**.

Now read the first 3 bits by copying **scratch** off to another variable and masking & ((1<<3) – 1), giving the 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. Scratch now looks like:

```
zzzzzzzzzzzzzzzzzzz
```

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. Or the work from memory into **scratch** after shifting left by **scratch_bits** (19). Now we have all the bits necessary for z in **scratch**:

```
zzzzzzzzzzzzzzzzzzzzzzzz
```

Read off 24 bits and shift **scratch** right by 24. s**cratch** is now all zeros. We're done.

Beyond Bitpacking

Bitpackers are an excellent optimization compared with reading and writing at a byte level, but reading and writing integer values into a packet by specifying the number of bits to read/write is not the most user friendly interface possible.

Consider this example:

```
const int MaxElements = 32;

struct PacketB
{
    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 forgets 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.

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

But be careful when array sizes aren't a power of two, because within the value serialized over the network lies the opportunity for an attacker to construct a packet that makes you trash memory!

In the example above **MaxElements** is 32, so **MaxElementBits** is 6. This seems fine because all valid values for **numElements:** [0,32] fit in 6 bits. The problem is that there are also additional values within 6 bits that are *outside* our array bounds: [33,63]. An attacker can use this fact to construct a malicious packet containing one of these values to make us trash 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 instead specify its valid range: [min,max]. This way if a value is outside of the expected range we can detect that and abort packet read.

I've implemented this abort using C++ exceptions in the past, but when I profiled this code, 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 this flag or return code on every read operation otherwise it's possible to get stuck in a memory trashing loop.

```
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. It's too

important to leave open to programmer error. You want it to be automatic.

Stay tuned for the next article where I show you a really neat way to do this in C++.

Next Article: Serialization Strategies

In the next article, I show you how to unify read and write operations into a single "Serialize" function that handles read and write at the same time without any extra runtime cost.

Please support my writing on patreon, and I'll write new articles faster, plus you get access to example source code for this article under the BSD 3-clause open source licence. **Thanks for your support!**

Original URL:
http://gafferongames.com/building-a-game-network-protocol/reading-and-writing-packets/