gafferongames.com

Packet Fragmentation and Reassembly

13 min read • original

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

In the previous article we discussed how to unify packet read and write into a single serialize function.

Now we actually want to put starting interestings things in our packets, but immediately we run into an interesting question: **How big should packets be?**

To answer this question you need to understand the concept of maximum transmission unit.

Maximum Transmission Unit (MTU)

When you send a packet over the internet is that it hops from one computer (router) to another in order to reach its destination. This is how a packet switched network operates. It's not like there's a direct wire connecting the between the source and destination IP addresses most of the time.

Each computer (router) along the route enforces a maximum packet size called the MTU. And if any router recieves a packet larger than its MTU, it has the option of a) fragmenting that packet at the IP level and passing it on, or b) *dropping that packet.*

So here's a situation that I see all the time. People write multiplayer game where the average packet size is quite small, lets say a few hundred bytes, but every now and then when a lot of stuff is happening in their game and a burst of packet loss occurs, packet get a lot larger than usual...

Suddenly, for a very small percentage of the player base, everything goes to hell.

Just last year (2015) I was talking with Alex Austin at Indiecade about networking in his game Sub Rosa. He had this strange networking bug he couldn't reproduce. For some reason, certain clients (one or two in his entire player base!) would randomly get disconnected from the game when a bunch of stuff was going on. Looking at the logs Alex said it seemed like packets just stopped getting through.

It sounded exactly like an MTU issue to me, and sure enough, when Alex limited his maximum packet size to a reasonable value the bug went away.

MTU in the real world

So what is a "reasonable packet size"?

On the Internet today (2016, IPv4) the MTU is typically 1500 bytes. Give or take a few bytes for UDP/IP packet header and you'll find that the typical number before packets start to get dropped or fragmented is somewhere around 1472.

You can try this out for yourself by running this command on MacOS X:

```
ping -g 56 -G 1500 -h 10 -D 8.8.4.4
```

On my machine it conks out around just below 1500 bytes as expected:

```
1404 bytes from 8.8.4.4: icmp_seq=134 ttl=56 time=11.945 ms
1414 bytes from 8.8.4.4: icmp_seq=135 ttl=56 time=11.964 ms
1424 bytes from 8.8.4.4: icmp_seq=136 ttl=56 time=13.492 ms
```

```
1434 bytes from 8.8.4.4: icmp_seq=137 ttl=56 time=13.652 ms
1444 bytes from 8.8.4.4: icmp_seq=138 ttl=56 time=133.241 ms
1454 bytes from 8.8.4.4: icmp_seq=139 ttl=56 time=17.463 ms
1464 bytes from 8.8.4.4: icmp_seq=140 ttl=56 time=12.307 ms
1474 bytes from 8.8.4.4: icmp_seq=141 ttl=56 time=11.987 ms
ping: sendto: Message too long
ping: sendto: Message too long
Request timeout for icmp_seq 142
```

Why 1500? That's the default MTU for MacOS X. It's also the default MTU on Windows. So now we have an upper bound for your packet size assuming you actually care about packets getting through to Windows and Mac boxes without IP level fragmentation or chance of being dropped: **1472 bytes.**

So what's the lower bound? MacOS X lets me set MTU values in range 1280 to 1500 so my first guess for a conservative lower bound on the IPv4 Internet today would be **1200 bytes**. Moving forward into IPv6 this is also a good value to use for conservative MTU, as any packet size <= 1280 bytes is guaranteed to get passed on without IP level fragmentation.

This lines up with numbers that I have seen throughout my career. In my experience games rarely try anything complicated like attempting to discover path MTU, they just assume a reasonably conservative MTU and roll with that. If a packet needs to be sent that is larger than the conservative MTU, the game protocol splits that packet up into fragments and re-assembles them on the other side.

That's exactly what I'm going to show you how to do in this article.

Fragment Packet Structure

Lets get started building up our own packet fragmentation and reassembly by deciding how we're going to represent fragment packets over the network. Ideally, we would like fragmented and non-fragmented packets to be compatible with the existing packet structure we've already built, with zero overhead in the network protocol when sending packets which are smaller than MTU.

Here's the packet structure from the end of the previous article:

```
~~[protocol id] (32 bits)~~ // not actually sent, but used to calc crc32
[crc32] (32 bits)
[packet type] (2 bits)
(variable length packet data according to packet type)
[end of packet serialize check] (32 bits)
```

We have three packet types in our example: A, B and C.

Lets make one of these packet types generate really large packets that go over MTU:

```
static const int MaxItems = 4096 * 4;

struct TestPacketB : public protocol2::Packet
{
    int numItems;
    int items[MaxItems];

    TestPacketB() : Packet( TEST_PACKET_B )
    {
        numItems = random_int( 0, MaxItems );
        for ( int i = 0; i < numItems; ++i )
            items[i] = random_int( -100, +100 );
    }

    template <typename Stream> bool Serialize( Stream & stream )
    {
        serialize_int( stream, numItems, 0, MaxItems );
        for ( int i = 0; i < numItems; ++i )
            serialize_int( stream, items[i], -100, +100 );
        return true;
    }
};
```

This may seem contrived but in the real world these situations really do occur. For example, if you have a strategy where you send all un-acked events from server to client and you hit a spike of reliable-ordered events and a burst of packet loss, you can easily end up with packets larger than MTU, even though your average packet size is quite small.

In most cases, it's a good idea to avoid this by implementing a strategy where you only include a subset of events or state updates into each packet in order to stay under a specified maximum packet size. This strategy works well in many cases… but there is one case where it doesn't: **delta encoding.**

Packets created by a delta encoder have a size proportional to the amount of state changed between the previous and the current state. If there are a lot of differences between the states then the delta will be large and there's simply nothing you can do about it. If this delta just happens to be bigger than MTU, tough luck, you still have to send it! So you can see that with delta encoding, you really can't limit to some maximum packet size that is below MTU, so a packet fragmentation and re-assembly strategy makes sense in that case.

Lets get back to packet structure. It's fairly common to add a sequence number at the header of each packet. This isn't anything complicated. It's just a packet number that increases with each packet sent. eg: 0,1,2,3. I like to use 16 bits for sequence numbers even though they wrap around in about 15 minutes @ 60 packets-per-second, it's very unlikely you will ever receive a ~15 minute old packet sent over the network and confuse it for a newer packet with the same number. If this is a concern for you, consider using 32 bit sequence numbers instead.

No matter what size sequence numbers you choose, they're useful for a bunch of things like implementing reliability and detecting and discarding out of order packets. Plus, we're definitely going to need a packet sequence number when fragmenting packets because we need some way to identify which packet a fragment belongs to.

So lets add a sequence number to our packet:

```
~~[protocol id] (32 bits)~~   // not actually sent, but used to calc crc
[crc32] (32 bits)
[sequence] (16 bits)
[packet type] (2 bits)
(variable length packet data according to packet type)
[end of packet serialize check] (32 bits)
```

Here's the interesting part. We could just add a bit **is_fragment** to the header, but then in the common case of non-fragmented packets you're wasting one bit that is always set to zero. Lame.

What I do instead is add a special "fragment packet" type:

```
enum TestPacketTypes
{
    PACKET_FRAGMENT = 0,     // RESERVED
    TEST_PACKET_A,
    TEST_PACKET_B,
    TEST_PACKET_C,
    TEST_PACKET_NUM_TYPES
};
```

And it just happens to be free because four packet types fit into the 2 bits we're already using for packet type. Now when a packet is read, if the packet type is zero then we know that a special fragmented packet layout follows the packet type, otherwise we run through the ordinary, non-fragmented read packet codepath.

Lets design what this fragment packet looks like. We'll allow a maximum of 256 fragments per-packet and have a fragment size of 1024 bytes. This gives a maximum packet size of 256k that we can send through this system, which should be enough for anybody, but please don't quote me on this.

With a small fixed size header, UDP header and IP header a fragment packet be well under the conservative MTU value of 1200. Plus, with 256 max fragments per-packet we can represent a fragment id in the range [0,255] and the total number of fragments per-packet [1,256] with 8 bits.

```
~~[protocol id] (32 bits)~~   // not actually sent, but used to calc crc
[crc32] (32 bits)
[sequence] (16 bits)
[packet type = 0] (2 bits)
[fragment id] (8 bits)
[num fragments] (8 bits)
[pad zero bits to nearest byte index]
<fragment data>
```

Notice that we pad bits up to the next byte before writing out the fragment data. Why do this? Two reasons: 1) it's faster to copy the fragment data into the packet via memcpy than bitpacking each byte, 2) we save a small amount of bandwidth by not sending the size of the fragment data. We infer it by subtracting the packet byte index at the start of fragment data from the total size of the packet.

Sending Packet Fragments

Sending packet fragments is easy. Just work out if the packet being sent is <= conservative MTU and if so, send the packet normally. Otherwise, calculate how many 1024 byte fragments it needs to be split into, construct those fragment packets and send them over the network. Fire and forget!

One consequence of this fire and forget approach is that if any fragment of that packet is lost then the entire packet is lost. It follows that if you have packet loss then sending a 256k packet as 256 fragments is probably not a very good idea.

It's a bad idea because the probability of dropping a packet increases significantly as the number of fragments. Not quite linearly, but in a moderately complicated way that you can read about here.

In short, to calculate the probability of losing a packet, you must calculate the probability of all fragments being delivered successfully and subtract that from one, giving you the probability that at least one fragment was dropped.

This gives you the probability that your fragmented packet will be dropped:

```
1 - ( probability of fragment being delivered ) ^ num_fragments
```

For example, if we send a non-fragmented packet over the network with 1% packet loss, there is a 1/100 chance that packet will be dropped, or, redundantly: 1 – (99/100) ^ 1 = 1/100 = **1%**

As the number of fragments increase, the chance of losing the packet also increases:

- Two fragments: 1 – (99/100) ^ 2 = **2%**
- Ten fragments: 1 – (99/100) ^ 10 = **9.5%**
- 100 fragments: 1 – (99/100) ^ 100 = **63.4%**
- 256 fragments: 1 – (99/100) ^ 256 = **92.4%**

So I recommend you take it easy with the number of fragments. It's best to use this strategy only for packets in the 2-4 fragment range, and only for time critical data that doesn't matter too much if it gets dropped like delta encoded state. It's **definitely not** a good idea to fire down a bunch of reliable-ordered events in a huge packet and rely on packet fragmentation and reassembly to save your ass.

Another typical use case for large packets is when a client joins a game. You usually want to send a large block of data down reliably to that client. Perhaps it represents the initial state of the world for late join. Whatever you do, don't send that block of data down using the fragmentation trick in this article. Instead, check out the technique in next article which lets you send large blocks of data quickly and reliably under packet loss by resending fragments until they are all received.

Receiving Packet Fragments

While sending fragmented packets is super easy, receiving them is actually pretty tricky.

The reason for this is that not only do we have to maintain a data structure to buffer and re-assemble fragments into packets, we also have to be particularly careful of somebody trying to crash us by sending malicious packets.

Here is the fragment packet we are receiving:

```
~~[protocol id] (32 bits)~~   // not actually sent, but used to calc crc
[crc32] (32 bits)
[sequence] (16 bits)
[packet type = 0] (2 bits)
[fragment id] (8 bits)
[num fragments] (8 bits)
[pad zero bits to nearest byte index]
<fragment data>
```

And here's a list of all the ways that I would try to attack your protocol to try to crash your server:

- Try to send out of bound fragments ids trying to get you to crash memory. eg: send fragments [0,255] in a packet that has just two fragments.
- Send packet n with some maximum fragment count of say 2, and then send more fragment packets belonging to the same packet n but with maximum fragments of 256 hoping that you didn't realize I widened the maximum number of fragments in the packet after the first one you received, and you trash memory.
- Send really large fragment packets with fragment data larger than 1k hoping to get you to trash memory as you try to copy that fragment data into the data structure, or blow memory budget trying to allocate fragments
- Continually send you fragments of maximum size (256/256 fragments) in hope that it I could make you allocate a bunch of memory and crash you out. Lets say you have a sliding window of

256 packets, 256 fragments per-packet max, and each fragment is 1k. That's 67,108,864 bytes or 64 mb per-client maximum allocated. Can I crash the server this way? Can I fragment your heap with a bunch of funny sized fragment packets? Only you as the owner of the server can know for sure. It depends on your memory budget and how you allocate memory to store fragments. If this is a problem, limit the number of fragments in the buffer at any point (across all packets) or consider reducing the maximum number of fragments per-packet. Consider statically allocating the data structure for fragments or using a pool to reduce memory fragmentation.

So you can see that on the receive side you are quite vulnerable and have to be extremely careful to check everything. Aside from this, it's reasonably straightforward: keep the fragments in a data structure, when all packets for a fragment arrive (keep a count) reassemble those fragments into a large packet and return the big packet to the receiver.

What sort of data structure makes sense here? Nothing fancy! It's something I like to call a sequence buffer. The key trick I'd like to share with you to make this data structure efficient is:

```
const int MaxEntries = 256;

struct SequenceBuffer
{
    bool exists[MaxEntries];
    uint16_t sequence[MaxEntries];
    Entry entries[MaxEntries];
};
```

There are a few things going on here. Firstly, structure-of-arrays (SoA) allows quick and cache efficient testing if a particular entry exists in the sequence buffer, even if the per-packet entry structure is is large (and for packet fragmentation and re-assembly it could very well be.)

So how do we use this data structure? When you have a packet fragment coming in it has a sequence number which identifies which packet the fragment belongs to. The sequence number keeps moving forward (with the exception of the wrap around event) so the key trick is that you hash the sequence number into a rolling index into the array like this:

```
 int index = sequence % MaxEntries;
```

Now you can quickly test at O(1) whether a particular entry exists by sequence number, and test if that entry matches the packet sequence number you would like to read or write. You need to test both the exists flag and sequence number, because all sequence numbers are valid numbers (eg. 0), and because the entry from your particular packet sequence number may exist, but belong to another sequence number from in the past (eg. some other sequence that resolves to the same index modulo MaxEntries).

So when the first fragment of a new packet comes in you hash the sequence to an index, discover that it does not exist yet, so you set exists[index] = 1 and the sequence[index] to match the packet sequence you are processing and store that fragment in that entry in the sequence buffer. The next time a fragment packet comes in, you have the same sequence number, and see that an entry for that sequence already exists, and that the sequence number for that entry matches your packet sequence number, so you accumulate the next fragment in that entry, and you keep doing this until eventually all fragments for that packet have been received.

And that's basically it at a high level. For a more complete explanation of this approach please refer to the example source code for this article. Click here to get the example source code for this article series.

Test Driven Development for Network Protocols

One thing I'd like to close this article out on. I feel like I'd be doing a disservice if I don't mention this approach to my readers.

Writing a custom network protocol is **hard**. So hard that I've done this from scratch at least 10 times but each time I manage to fuck it up in a new and interesting ways. You'd think I'd learn eventually but this stuff is complicated. You can't just write the code and expect it to work. You have to test it!

My strategy when writing low-level network protocol layer code is:

- Code defensively. Assert everywhere. These asserts will fire and they'll be important clues you need when something goes wrong.
- Add functional tests and make sure stuff is working as you are writing. Put your code through its paces at a basic level as you write it and make sure it's working as you build it up. Think hard about cases that make sense that need to be properly handled and add tests for those.
- Just adding a bunch of functional tests is not enough. There are of course cases you didn't think of! Now you have to get really mean. I call this soak testing and I've never, not even once, have coded a network protocol that hasn't subsequently had problems found in it by soak testing. When soak testing just loop forever and just do a mix of random stuff that puts your system through its paces, eg. random length packets in this case with a huge amount of packet loss, out of order and duplicates through a packet simulator. Your soak test passes when it runs overnight and doesn't hang or assert.
- If you find anything wrong with soak testing. You may need to go back and add detailed logs to the soak test to work out how you got to the failure case. Once you know what's going on, STOP. Don't fix it immediately and just run the soak test again. That's stupid. Instead, add a unit test that reproduces that problem you are trying to fix, verify that test reproduces the problem, and that the problem goes away with your fix. Only after this, go back to the soak test and make sure they run overnight. This way the unit tests document the correct behavior of your system and can quickly be run in future to make sure you don't break this thing moving forward when you make other changes.

That's my process and it seems to work pretty well. If you are doing low-level network protocol design, the rest of your game depends on this code. You need to be absolutely sure it works before you build on it, otherwise it's like a stack of cards. Game neworking is hard enough without having a suspicion that that your low-level network protocol might not be working properly or has bugs in it. So make sure you know it works!

I hope you're enjoyed the writing in this series so far. 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!**

Up next: Sending Large Blocks of Data

Read on for the next article in this series where I show you how to send large blocks of data quickly and reliably over UDP even under high packet loss, without dropping the block if a fragment is lost.

Original URL:
http://gafferongames.com/building-a-game-network-protocol/packet-fragmentation-and-reassembly/