Packet Fragmentation and Reassembly

How to send and receive packets larger than MTU

Posted by Glenn Fiedler (http://web.archive.org/web/20181107181428/https://gafferongames.com/about) on Tuesday, September 6, 2016

Introduction

Hi, I'm Glenn Fiedler (http://web.archive.org/web/20181107181428/https://gafferongames.com/about) and welcome to **Building a Game Network Protocol (http://web.archive.org/web/20181107181428/https://gafferongames.com/categories/building-a-game-network-protocol/)**.

In the previous article (http://web.archive.org/web/20181107181428/https://gafferongames.com/post/serialization_strategies/) we discussed how to unify packet read and write into a single serialize function and added a bunch of safety features to packet read.

Now we are ready to start putting interesting things in our packets and sending them over the network, but immediately we run into an interesting question: *how big should our packets be?*

To answer this question properly we need a bit of background about how packets are actually sent over the Internet.

Background

Perhaps the most important thing to understand about the internet is that there's no direct connection between the source and destination IP address. What actually happens is that packets hop from one computer to another to reach their destination.

Each computer along this route enforces a maximum packet size called the maximum transmission unit, or MTU. According to the IP standard, if any computer recieves a packet larger than its MTU, it has the option of a) fragmenting that packet, or b) dropping the packet.

So here's how this usually goes down. People write a 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, packets get a lot larger than usual, going above MTU for the route, and suddenly all packets start getting dropped!

Just last year (2015) I was talking with Alex Austin at Indiecade about networking in his game Sub Rosa (http://web.archive.org/web/20181107181428/http://subrosagame.com/). He had this strange networking bug he couldn't reproduce. For some reason, players would randomly get disconnected from the game, but only

This 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's a reasonable maximum packet size?

On the Internet today (2016, IPv4) the real-world MTU is 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 a chance of being dropped: **1472 bytes**.

So what's the lower bound? Unfortunately for the routers in between your computer and the destination the IPv4 standard says **576**. Does this mean we have to limit our packets to 400 bytes or less? In practice, not really.

MacOS X lets me set MTU values in range 1280 to 1500 so considering packet header overhead, my first guess for a conservative lower bound on the IPv4 Internet today would be **1200 bytes**. Moving forward, in IPv6 this is also a good value, as any packet of 1280 bytes or less is guaranteed to get passed on without IP level fragmentation.

This lines up with numbers that I've 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, something like 1000 to 1200 bytes of payload data. If a packet larger than this needs to be sent, it's split up into fragments by the game protocol and re-assembled on the other side.

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

Fragment Packet Structure

Let's get started with implementation.

Ideally, we would like fragmented and non-fragmented packets to be compatible with the existing packet structure we've already built, with as little overhead as possible in the common case when we are sending packets smaller than MTU.

Here's the packet structure from the previous article:

```
~~[protocol id] (64 bits)~~ // not actually sent, but used to calc crc32
[crc32] (32 bits)
[packet type] (2 bits for 3 distinct packet types)
(variable length packet data according to packet type)
[end of packet serialize check] (32 bits)
```

In our protocol we have three packet types: A, B and C.

Let's make one of these packet types generate really large packets:

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

struct TestPacketB : public 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 somewhat contrived but 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 burst of packet loss, you can easily end up with packets larger than MTU, even though your average packet size is quite small.

Another common case is delta encoded snapshots in a first person shooter. Here packet size is proportional to the amount of state changed between the baseline and current snapshots for each client. If there are a lot of differences between the snapshots the delta packet is large and there's nothing you can do about it except break it up into fragments and re-assemble them on the other side.

Getting back to packet structure. It's fairly common to add a sequence number at the header of each packet. This is just a packet number that increases with each packet sent. I like to use 16 bits for sequence numbers even though they wrap around in about 15 minutes @ 60 packets-per-second, because it's extremely unlikely that a packet will be delivered 15 minutes late.

Sequence numbers are useful for a bunch of things like acks, reliability and detecting and discarding out of order packets. In our case, we're going to use the sequence number to identify which packet a fragment belongs to:

```
[packet type] (2 bits)
(variable length packet data according to packet type)
[end of packet serialize check] (32 bits)
```

Here's the interesting part. Sure 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.

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 2 bits. Now when a packet is read, if the packet type is zero we know it's a fragment packet, 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 crc32
[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 fragment data into the packet via memcpy than bitpacking each byte, and 2) we can now save a small amount of bandwidth by inferring the fragment size by subtracting the start of the fragment data from the total size of the packet.

Sending Packet Fragments

Sending packet fragments is *easy*. For any packet larger than conservative MTU, simply calculate how many 1024 byte fragments it needs to be split into, and send those fragment packets over the network. Fire and forget!

One consequence of this 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 not a very good idea, because the probability of dropping a packet increases significantly as the number of fragments increases. Not quite linearly, but in an interesting way that you can read more about here (http://web.archive.org/web/20181107181428/https://www.fourmilab.ch/rpkp/experiments/statistics.html).

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 naturally a $^1/_{100}$ chance the packet will be dropped.

As the number of fragments increase, packet loss is amplified:

- 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. 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 initially joins a game. Here you usually want to send a large block of data down reliably to that client, for example, representing the initial state of the world for late join. Whatever you do, don't send that block of data down using the fragmentation and re-assembly technique in this article.

Instead, check out the technique in next article (http://web.archive.org/web/20181107181428/https://gafferongames.com/post/sending-large-blocks-of-data.html) which handles packet loss by resending fragments until they are all received.

Receiving Packet Fragments

It's time to implement the code that receives and processed packet fragments. This is a bit tricky because we have to be particularly careful of somebody trying to attack us with malicious packets.

Here's a list of all the ways I can think of to attack the protocol:

- 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 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 64 mb per-client.

because the heap becomes fragmented.

Aside from these concerns, implementation is reasonably straightforward: store received fragments somewhere and when all fragments arrive for a packet, reassemble them into the original packet and return that to the user.

Data Structure on Receiver Side

The first thing we need is some way to store fragments before they are reassembled. My favorite data structure is something I call a *sequence buffer*:

```
const int MaxEntries = 256;

struct SequenceBuffer
{
    uint32_t sequence[MaxEntries];
    Entry entries[MaxEntries];
};
```

Indexing into the arrays is performed with modulo arithmetic, giving us a fast O(1) lookup of entries by sequence number:

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

A sentinel value of 0xFFFFFFFF is used to represent empty entries. This value cannot possibly occur with 16 bit sequence numbers, thus providing us with a fast test to see if an entry exists for a given sequence number, without an additional branch to test if that entry exists.

This data structure is used as follows. When the first fragment of a new packet comes in, the sequence number is mapped to an entry in the sequence buffer. If an entry doesn't exist, it's added and the fragment data is stored in there, along with information about the fragment, eg. how many fragments there are, how many fragments have been received so far, and so on.

Each time a new fragment arrives, it looks up the entry by the packet sequence number. When an entry already exists, the fragment data is stored and number of fragments received is incremented. Eventually, once the number of fragments received matches the number of fragments in the packet, the packet is reassembled and delivered to the user.

Since it's possible for old entries to stick around (potentially with allocated blocks), great care must be taken to clean up any stale entries when inserting new entries in the sequence buffer. These stale entries correspond to packets that didn't receive all fragments.

And that's basically it at a high level. For further details on this approach please refer to the example source code for this article.

Click here to get the example source code for this article series (http://web.archive.org/web/20181107181428/https://www.patreon.com/gafferongames).

Test Driven Development

One thing I'd like to close this article out on.

this stuff is complicated. You can't just write low-level netcode and expect it to just work.

You have to test it!

My strategy when testing low-level netcode is as follows:

- 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 it. 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 the essential cases that need to be property handled and add tests that cover them.

- But 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.

- Instead, add a unit test that reproduces that problem you are trying to fix, verify your test reproduces the problem, and that it 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.

- Add a bunch of logs. High level errors, info asserts showing an overview of what is going on, but also low-level warnings and debug logs that show what went wrong after the fact. You're going to need these logs to diagnose issues that don't occur on your machine. Make sure the log level can be adjusted dynamically.

- Implement network simulators and make sure code handles the worst possible network conditions imaginable. 99% packet loss, 10 seconds of latency and +/- several seconds of jitter. Again, you'll be surprised how much this uncovers. Testing is the time where you want to uncover and fix issues with bad network conditions, not the night before your open beta.

- Implement fuzz tests where appropriate to make sure your protocol doesn't crash when processing random packets. Leave fuzz tests running overnight to feel confident that your code is reasonably secure against malicious packets and doesn't crash.

- Surprisingly, I've consistently found issues that only show up when I loop the set of unit tests over and over, perhaps these issues are caused by different random numbers in tests, especially with the network simulator being driven by random numbers. This is a great way to take a rare test that fails once every few days and make it fail every time. So before you congratulate yourself on your tests passing 100%, add a mode where your unit tests can be looped easily, to uncover such errors.

multiple platforms as you develop, otherwise it's pretty painful fixing all these at the end.

- Think about how people can attack the protocol. Implement code to defend against these attacks. Add functional tests that mimic these attacks and make sure that your code handles them correctly.

This is my process and it seems to work pretty well. If you are writing a low-level network protocol, the rest of your game depends on this code working correctly. You need to be absolutely sure it works before you build on it, otherwise it's basically a stack of cards.

In my experience, game neworking is hard enough without having suspicions that that your low-level network protocol has bugs that only show up under extreme network conditions. That's exactly where you need to be able to trust your code works correctly. **So test it!**

NEXT ARTICLE: Sending Large Blocks of Data
(http://web.archive.org/web/20181107181428/https://gafferongames.com/post/sending_large_blocks_of_data/

● (http://web.archive.org/web/20181107181428/https://www.linkedin.com/in/glennfiedler/)
● (http://web.archive.org/web/20181107181428/https://twitter.com/gafferongames)
● (http://web.archive.org/web/20181107181428/https://github.com/gafferongames)