Sending Large Blocks of Data

How to send blocks quickly and reliably over UDP

Posted by Glenn Fiedler (http://web.archive.org/web/20181107181529/https://gafferongames.com/about) on Monday, September 12, 2016

Introduction

Hi, I'm Glenn Fiedler (http://web.archive.org/web/20181107181529/https://gafferongames.com/about) and welcome to **Building a Game Network Protocol (http://web.archive.org/web/20181107181529/https://gafferongames.com/categories/building-a-game-network-protocol/)**.

In the previous article (http://web.archive.org/web/20181107181529/https://gafferongames.com/post/packet_fragmentation_and_rea
implemented packet fragmentation and reassembly so we can send packets larger than MTU.

This approach works great when the data block you're sending is time critical and can be dropped, but in other cases you need to send large blocks of quickly and reliably over packet loss, and you need the data to get through.

In this situation, a different technique gives much better results.

Background

It's common for servers to send large block of data to the client on connect, for example, the initial state of the game world for late join.

Let's assume this data is 256k in size and the client needs to receive it before they can join the game. The client is stuck behind a load screen waiting for the data, so obviously we want it to be transmitted as quickly as possible.

If we send the data with the technique from the previous article, we get *packet loss amplification* because a single dropped fragment results in the whole packet being lost. The effect of this is actually quite severe. Our example block split into 256 fragments and sent over 1% packet loss now has a whopping 92.4% chance of being dropped!

Since we just need the data to get across, we have no choice but to keep sending it until it gets through. On average, we have to send the block 10 times before it's received. You may laugh but this actually happened on a AAA game I worked on!

fixing a bunch of players stalling out on connect, while continuing to send time critical data (snapshots) via packet fragmentation and reassembly.

Chunks and Slices

In this new system blocks of data are called *chunks*. Chunks are split up into *slices*. This name change keeps the chunk system terminology (chunks/slices) distinct from packet fragmentation and reassembly (packets/fragments).

The basic idea is that slices are sent over the network repeatedly until they all get through. Since we are implementing this over UDP, simple in concept becomes a little more complicated in implementation because have to build in our own basic reliability system so the sender knows which slices have been received.

This reliability gets quite tricky if we have a bunch of different chunks in flight, so we're going to make a simplifying assumption up front: we're only going to send one chunk over the network at a time. This doesn't mean the sender can't have a local send queue for chunks, just that in terms of network traffic there's only ever one chunk *in flight* at any time.

This makes intuitive sense because the whole point of the chunk system is to send chunks reliably and in-order. If you are for some reason sending chunk 0 and chunk 1 at the same time, what's the point? You can't process chunk 1 until chunk 0 comes through, because otherwise it wouldn't be reliable-ordered.

That said, if you dig a bit deeper you'll see that sending one chunk at a time does introduce a small trade-off, and that is that it adds a delay of RTT between chunk n being received and the send starting for chunk n+1 from the receiver's point of view.

This trade-off is totally acceptable for the occasional sending of large chunks like data sent once on client connect, but it's definitely *not* acceptable for data sent 10 or 20 times per-second like snapshots. So remember, this system is useful for large, infrequently sent blocks of data, not for time critical data.

Packet Structure

There are two sides to the chunk system, the **sender** and the **receiver**.

The sender is the side that queues up the chunk and sends slices over the network. The receiver is what reads those slice packets and reassembles the chunk on the other side. The receiver is also responsible for communicating back to the sender which slices have been received via acks.

The netcode I work on is usually client/server, and in this case I usually want to be able to send blocks of data from the server to the client *and* from the client to the server. In that case, there are two senders and two receivers, a sender on the client corresponding to a receiver on the server and vice-versa.

Think of the sender and receiver as end points for this chunk transmission protocol that define the direction of flow. If you want to send chunks in a different direction, or even extend the chunk sender to support peer-to-peer, just add sender and receiver end points for each direction you need to send chunks.

Traffic over the network for this system is sent via two packet types:

- **Slice packet** - contains a slice of a chunk up to 1k in size.
- **Ack packet** - a bitfield indicating which slices have been received so far.

maximum of 1k and there is a maximum of 256 slices per-chunk, therefore the largest data you can send over the network with this system is 256k.

```
const int SliceSize = 1024;
const int MaxSlicesPerChunk = 256;
const int MaxChunkSize = SliceSize * MaxSlicesPerChunk;

struct SlicePacket : public protocol2::Packet
{
    uint16_t chunkId;
    int sliceId;
    int numSlices;
    int sliceBytes;
    uint8_t data[SliceSize];

    template &lt;typename Stream&gt; bool Serialize( Stream &amp; stream )
    {
        serialize_bits( stream, chunkId, 16 );
        serialize_int( stream, sliceId, 0, MaxSlicesPerChunk - 1 );
        serialize_int( stream, numSlices, 1, MaxSlicesPerChunk );
        if ( sliceId == numSlices - 1 )
        {
            serialize_int( stream, sliceBytes, 1, SliceSize );
        }
        else if ( Stream::IsReading )
        {
            sliceBytes = SliceSize;
        }
        serialize_bytes( stream, data, sliceBytes );
        return true;
    }
};
```

There are two points I'd like to make about the slice packet. The first is that even though there is only ever one chunk in flight over the network, it's still necessary to include the chunk id (0,1,2,3, etc...) because packets sent over UDP can be received out of order.

Second point. Due to the way chunks are sliced up we know that all slices except the last one must be SliceSize (1024 bytes). We take advantage of this to save a small bit of bandwidth sending the slice size only in the last slice, but there is a trade-off: the receiver doesn't know the exact size of a chunk until it receives the last slice.

The other packet sent by this system is the ack packet. This packet is sent in the opposite direction, from the receiver back to the sender. This is the reliability part of the chunk network protocol. Its purpose is to lets the sender know which slices have been received.

```
struct AckPacket : public protocol2::Packet
{
    uint16_t chunkId;
    int numSlices;
    bool acked[MaxSlicesPerChunk];

    bool Serialize( Stream &amp; stream )
    {
        serialize_bits( stream, chunkId, 16 );
        serialize_int( stream, numSlices, 1, MaxSlicesPerChunk );
        for ( int i = 0; i &lt; numSlices; ++i )
        {
            serialize_bool( stream, acked[i] ); return true; } };
        }
    }
};
```

Acks are short for 'acknowledgments'. So an ack for slice 100 means the receiver is *acknowledging* that it has received slice 100. This is critical information for the sender because not only does it let the sender determine when all slices have been received so it knows when to stop, it also allows the sender to use bandwidth more efficiently by only sending slices that haven't been acked.

going to get through. You certainly don't want a desync between the sender and the receiver regarding which slices are acked.

So we need some reliability for acks, but we don't want to implement an *ack system for acks* because that would be a huge pain in the ass. Since the worst case ack bitfield is just 256 bits or 32 bytes, we just send the entire state of all acked slices in each ack packet. When the ack packet is received, we consider a slice to be acked the instant an ack packet comes in with that slice marked as acked and locally that slice is not seen as acked yet.

This last step, biasing in the direction of non-acked to ack, like a fuse getting blown, means we can handle out of order delivery of ack packets.

Sender Implementation

Let's get started with the implementation of the sender.

The strategy for the sender is:

- Keep sending slices until all slices are acked
- Don't resend slices that have already been acked

We use the following data structure for the sender:

```
class ChunkSender
{
    bool sending;
    uint16_t chunkId;
    int chunkSize;
    int numSlices;
    int numAckedSlices;
    int currentSliceId;
    bool acked[MaxSlicesPerChunk];
    uint8_t chunkData[MaxChunkSize];
    double timeLastSent[MaxSlicesPerChunk];
};
```

As mentioned before, only one chunk is sent at a time, so there is a 'sending' state which is true if we are currently sending a chunk, false if we are in an idle state ready for the user to send a chunk. In this implementation, you can't send another chunk while the current chunk is still being sent over the network. If you don't like this, stick a queue in front of the sender.

Next, we have the id of the chunk we are currently sending, or, if we are not sending a chunk, the id of the next chunk to be sent, followed by the size of the chunk and the number of slices it has been split into. We also track, per-slice, whether that slice has been acked, which lets us count the number of slices that have been acked so far while ignoring redundant acks. A chunk is considered fully received from the sender's point of view when numAckedSlices == numSlices.

We also keep track of the current slice id for the algorithm that determines which slices to send, which works like this. At the start of a chunk send, start at slice id 0 and work from left to right and wrap back around to 0 again when you go past the last slice. Eventually, you stop iterating across because you've run out of bandwidth to send slices. At this point, remember our current slice index via current slice id so you can pick up from where you left off next time. This last part is important because it distributes sends across all slices, not just the first few.

as you walk across slices and consider each slice you want to send, estimate roughly how many bytes the slice packet will take eg: roughly slice bytes + some overhead for your protocol and UDP/IP header. Then compare the amount of bytes required vs. the available bytes you have to send in your bandwidth budget. If you don't have enough bytes accumulated, stop. Otherwise, subtract the bytes required to send the slice and repeat the process for the next slice.

Where does the available bytes in the send budget come from? Each frame before you update the chunk sender, take your target bandwidth (eg. 256kbps), convert it to bytes per-second, and add it multiplied by delta time (dt) to an accumulator.

A conservative send rate of 256kbps means you can send 32000 bytes per-second, so add 32000 * dt to the accumulator. A middle ground of 512kbit/sec is 64000 bytes per-second. A more aggressive 1mbit is 125000 bytes per-second. This way each update you *accumulate* a number of bytes you are allowed to send, and when you've sent all the slices you can given that budget, any bytes left over stick around for the next time you try to send a slice.

One subtle point with the chunk sender and is that it's a good idea to implement some minimum resend delay per-slice, otherwise you get situations where for small chunks, or the last few slices of a chunk that the same few slices get spammed over the network.

For this reason we maintain an array of last send time per-slice. One option for this resend delay is to maintain an estimate of RTT and to only resend a slice if it hasn't been acked within RTT * 1.25 of its last send time. Or, you could just resend the slice it if it hasn't been sent in the last 100ms. Works for me!

Kicking it up a notch

Do the math you'll notice it still takes a long time for a 256k chunk to get across:

- 1mbps = 2 seconds
- 512kbps = 4 seconds
- 256kbps = **8 seconds :(**

Which kinda sucks. The whole point here is quickly and reliably. Emphasis on *quickly*. Wouldn't it be nice to be able to get the chunk across faster? The typical use case of the chunk system supports this. For example, a large block of data sent down to the client immediately on connect or a block of data that has to get through before the client exits a load screen and starts to play. You want this to be over as quickly as possible and in both cases the user really doesn't have anything better to do with their bandwidth, so why not use as much of it as possible?

One thing I've tried in the past with excellent results is an initial burst. Assuming your chunk size isn't so large, and your chunk sends are infrequent, I can see no reason why you can't just fire across the entire chunk, all slices of it, in separate packets in one glorious burst of bandwidth, wait 100ms, and then resume the regular bandwidth limited slice sending strategy.

Why does this work? In the case where the user has a good internet connection (some multiple of 10mbps or greater…), the slices get through very quickly indeed. In the situation where the connection is not so great, the burst gets buffered up and *most* slices will be delivered as quickly as possible limited only by the amount bandwidth available. After this point switching to the regular strategy at a lower rate picks up any slices that didn't get through the first time.

at almost any cost. It's a TCP thing. Normally, I hate this because it induces latency in packet delivery and messes up your game packets which you want delivered as quickly as possible, but in this case it's good behavior because the player really has nothing else to do but wait for your chunk to get through.

Just don't go too overboard with the spam or the congestion will persist after your chunk send completes and it will affect your game for the first few seconds. Also, make sure you increase the size of your OS socket buffers on both ends so they are larger than your maximum chunk size (I recommend at least double), otherwise you'll be dropping slices packets before they even hit the wire.

Finally, I want to be a responsible network citizen here so although I recommend sending all slices once in an initial burst, it's important for me to mention that I think this really is only appropriate, and only really *borderline appropriate* behavior for small chunks in the few 100s of k range in 2016, and only when your game isn't sending anything else that is time-critical.

Please don't use this burst strategy if your chunk is really large, eg: megabytes of data, because that's way too big to be relying on the kindness of strangers, AKA. the buffers in the routers between you and your packet's destination. For this it's necessary to implement something much smarter. Something adaptive that tries to send data as quickly as it can, but backs off when it detects too much latency and/or packet loss as a result of flooding the connection. Such a system is outside of the scope of this article.

Receiver Implementation

Now that we have the sender all sorted out let's move on to the reciever.

As mentioned previously, unlike the packet fragmentation and reassembly system from the previous article, the chunk system only ever has one chunk in flight.

This makes the reciever side of the chunk system much simpler:

```
class ChunkReceiver
{
    bool receiving;
    bool readyToRead;
    uint16_t chunkId;
    int chunkSize;
    int numSlices;
    int numReceivedSlices;
    bool received[MaxSlicesPerChunk];
    uint8_t chunkData[MaxChunkSize];
};
```

We have a state whether we are currently 'receiving' a chunk over the network, plus a 'readyToRead' state which indicates that a chunk has received all slices and is ready to be popped off by the user. This is effectively a minimal receive queue of length 1. If you don't like this, of course you are free to add a queue.

In this data structure we also keep track of chunk size (although it is not known with complete accuracy until the last slice arrives), num slices and num received slices, as well as a received flag per-slice. This per-slice received flag lets us discard packets containing slices we have already received, and count the number of slices received so far (since we may receive the slice multiple times, we only increase this count the first time we receive a particular slice). It's also used when generating ack packets. The chunk receive is completed from the receiver's point of view when numReceivedSlices == numSlices.

So what does it look like end-to-end receiving a chunk?

correct position, numSlices is set to the value in that packet, numReceivedSlices is incremented from 0 -> 1, and the received flag in the array entry corresponding to that slice is set to true.

As the remaining slice packets for the chunk come in, each of them are checked that they match the current chunk id and numSlices that are being received and are ignored if they don't match. Packets are also ignored if they contain a slice that has already been received. Otherwise, the slice data is copied into the correct place in the chunkData array, numReceivedSlices is incremented and received flag for that slice is set to true.

This process continues until all slices of the chunk are received, at which point the receiver sets receiving to 'false' and 'readyToRead' to true. While 'readyToRead' is true, incoming slice packets are discarded. At this point, the chunk receive packet processing is performed, typically on the same frame. The caller checks 'do I have a chunk to read?' and processes the chunk data. All chunk receive data is cleared back to defaults, except chunk id which is incremented from 0 -> 1, and we are ready to receive the next chunk.

Conclusion

The chunk system is simple in concept, but the implementation is certainly not. I encourage you to take a close look at the source code (http://web.archive.org/web/20181107181529/http://www.patreon.com/gafferongames) for this article for further details.

NEXT ARTICLE: Reliable Ordered Messages (http://web.archive.org/web/20181107181529/https://gafferongames.com/post/reliable_ordered_messages/)

● (http://web.archive.org/web/20181107181529/https://www.linkedin.com/in/glennfiedler/)

● (http://web.archive.org/web/20181107181529/https://twitter.com/gafferongames)

● (http://web.archive.org/web/20181107181529/https://github.com/gafferongames)