gafferongames.com

Sending Large Blocks of Data

18 min read • original

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

In the previous article we implemented packet fragmentation and reassembly at the game protocol level.

Now in this article we continue on our quest to build a professional grade game network protocol by exploring an alternative technique for sending large blocks of data over UDP.

This alternative technique seems similar to packet fragmentation and reassembly at first, but its implementation is quite different. This difference in implementation is intended to address a key weakness of packet fragmentation and reassembly – that the loss of one fragment results in the loss of the entire packet. This behavior is bad because it *amplifies* packet loss as the number of fragments increase. By the time you hit large blocks the amplification is so pronounced that a 256k block sent over 1% packet loss has a 92.4% chance of being dropped. You'd need to send that block 10 times on average for it to get through!

Obviously, this is totally unacceptable if you need to send large blocks of data quickly and reliably over networks with packet loss, like you know, ***THE INTERNET***. Some common examples where you need to do

this include: a large block of data that needs to get down to a client on initial join (perhaps the initial state of the world), the baseline for the start of a delta encoder, or a block of data that a client is waiting on to progress past a load screen in a multiplayer game.

It's very important in all these situations not only to handle packet loss gracefully, but also to take advantage of available bandwidth and send the block of data as quickly as possible.

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

Chunks and Slices

Lets get started with basic terminology. In this new system the large blocks of data are called 'chunks' and the fragments they are split up into are called 'slices'. This name change keeps the chunk system terminology (slices) distinct from packet fragmentation and reassembly (fragments), which is something I think is quite important because these systems solve different problems and there's no reason why you can't use both in the same network protocol. In fact, I often combine the two using packet fragmentation and reassembly for time critical state delta packets, while using the chunk system to send down the (much larger) initial state when a client joins the game.

The basic idea of the chunk system, and it's really not complicated at all, is that chunks are split up into slices and those slices are sent over the network repeatedly until they all get through. Of course, since we are implementing this over UDP with packet loss, out of order packet delivery and duplicate packets, simple in concept becomes a little more complicated in implementation, because we have to build in our own basic reliability system over UDP so the sender knows which slices have been received.

This reliability gets quite tricky if we have a bunch of different chunks in flight (like we did with packet fragmentation and reassembly) 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 actual network traffic there's only ever one chunk in flight.

This makes a 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, otherwise it wouldn't be reliable-ordered. That being 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 (eg. chunk sent on client connect, chunk sent once per-level load...), but it's definitely not acceptable for chunks sent 10 or 20 times per-second. So I hope you can see what this system is designed for vs. what it is not.

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 them back into a 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 that 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, one of each on each side. So basically, 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

another direction, or even extend the chunk sender to support peer-to-peer, just add more sender and receiver end points for each direction you want to send chunks.

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

- **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.

The slice packet is sent from the sender to the receiver. It is the payload that we use to get the chunk data across the network and is designed so that each packet fits neatly under a conservative MTU of 1200 bytes. Each slice is a maximum of 1k and there is a maximum of 256 slices per-chunk, so the largest data you can send over the network with this system is 256k (you could increase this if you wish by bumping the max # of slices). I recommend leaving slice size at 1k for MTU reasons.

```
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 <typename Stream> bool Serialize( Stream & 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 a chunk id (eg. 0,1,2,3, etc...) because packets sent over UDP can be received out of order. This way if a slice packet comes in corresponding to an old chunk, eg: you are receiving chunk 2, but an old packet containing a slice from chunk 1 comes in, you can reject that packet instead of splicing it's data into chunk 2 and corrupting it.

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 by only sending the slice size in the last slice, but there is a small trade-off as a result – the receiver doesn't know the exact size of the chunk in bytes until it receives the last slice.

Moving forward the other packet sent by this system is the ack packet. This packet is sent in the other direction, from the receiver back to the sender, and 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 & stream )
    {
        serialize_bits( stream, chunkId, 16 );
        serialize_int( stream, numSlices, 1, MaxSlicesPerChunk );
        for ( int i = 0; i < 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 target bandwidth more efficiently by only resending slices that have not been acked yet.

Looking a bit deeper into the ack packet, at first glance it might seems a bit redundant that we send acks for all slices in every packet. If you are not used to designing UDP protocols this may seems quite strange. Why are we doing this? Well, the reason is that ack packets are sent over UDP so they can be lost and received out of order. So it's not just slice packets that need reliability, ack packets need reliability too. You certainly don't want a desync between the sender and the receiver as to 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, the simplest approach here is also the best. Send the entire state of all acked slices in each ack packet, and when the ack packet is received, consider a slice to be acked the instant when an ack packet comes in with that slice marked as acked and locally it is not seen as acked yet. This last step, biasing in the direction of non-acked to ack, like a fuse getting blown, and ignoring any ack packet that comes in and says for example that a slice n that the sender already thinks is acked is not acked (old ack packet) means we also handle out of order delivery of acks packets.

Basic Sender Implementation

Now that we have the basic idea behind the system, lets 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 previously, 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; you have to wait for the current chunk to finish sending before you can send another. If you don't like this, stick a send queue in front of the chunk sender if you wish.

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, plus 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 avoid resending slices that have already been received and count the number of slices that have been acked so far while ignoring redundant acks. A chunk is 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 we go past the last slice (num slices – 1). Eventually, we stop iterating across because we've run out of bandwidth to send slices. At this point we remember our current slice index via current slice id so we can pick up from where we left off next time. This is important because it distributes sends across all slices in a chunk, not just the first few.

Now lets discuss bandwidth limiting. Obviously you don't just blast slices out continuously as you'd flood the connection in no time, so how do we limit the sender bandwidth? My implementation works something like this: 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 this slice and repeat the same process for the next slice.

Where does the available bytes to 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, while bytes left over when you don't have enough budget to send another slice stick around and are applied the next time you send a chunk.

One subtle point with the chunk sender and that 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 unnecessarily. It is 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 say 'fuck it' and resend the slice it if it hasn't been sent in the last 100ms. Works for me!

Kicking the sender up a notch

If you 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.

This seems a bit risky so let me explain. In the case where the user can't quite support this bandwidth what you're relying on here is that routers on the Internet *strongly prefer* to buffer packets rather than discard them at almost any cost. This is 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 at the same time. For example, please use the conservative strategy if you are sending a block of data down while the user is playing your game otherwise you risk affecting their playing experience by inducing additional latency and/or packet loss.

Please don't use this burst strategy if your chunk is large eg. megabytes to 10s of megabytes range, 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 sustained high throughput delivery of *really* large blocks of data it's necessary to implement something smarter. Something adaptive that tries to send data as quickly as it can, but backs off as 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 out of the way lets move on to the reciever. As mentioned previously, unlike the packet fragmentation and reassembly system from the previous article, the chunk system there is only ever one chunk in flight.

This makes the reciever side of the chunk system much simpler, as you can see below:

```
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 immediately pop the chunk data off the receiver and stick it in a real receive queue instead.

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 lets us 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 is also used when generating ack packets. The chunk receive is completed when numReceivedSlices == numSlices.

What does it look like end-to-end receiving a chunk?

First, the receiver sets up set to start at chunk 0. When the a slice packet comes in over the network matching the current chunk id (0), 'receiving' flips from false to true, data for that first slice is inserted into 'chunkData' at the correct position, numSlices is set to the value in that packet, numReceivedSlices is incremented from 0 -> 1, and the received flag 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 slice id and numSlices that are being received and discarded if they don't match. Packets are also discarded 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 all incoming slice packets are discarded. At this point, typically very shortly after the chunk receive packet processing is performed 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 0 -> 1, and we are ready to receive the next chunk.

Acks and the Importance of Soak Testing

At first glance the ack system seems really simple:

- Keep trace of slices that have been received
- When a slice packet is received, reply with an ack packet containing all acked slices

This seems fairly straightforward to implement, but like most things over UDP there are some subtle points that make it a bit tricky when packet loss gets involved.

A naive implementation for acks might work like this. Each time a slice is received, reply with an ack packet containing all acks (including the slice that was just received). This works logically, but it leaves the chunk protocol open for a malicious sender to exploit it as a DDoS tool. How? Well, if you 1-1 reply to each slice sent to you with an ack, and the sender is able to construct a *tiny* slice packet and you respond with an ack packet that's larger than the slice packet sent to you, you have provided somebody with a way to amplify their DDoS attack.

Now maybe I'm being paranoid here (I definitely am) but you can guard against DDoS amplification by in general, never designing your protocol such that a packet received has a one-to-one mapping to a another packet sent in response. eg. Don't make it so that if somebody sends you 1000 slice packets you reply with 1000 ack packets. Sent one ack packet instead, and only one ack packet sent at a rate of one packet every 50 or 100ms instead. This way abuse of your UDP protocol for DDoS amplification is not possible.

There are other ways this ack system tends to go wrong and these all tend to manifest as 'send hangs'. In other words, the receiver knows the chunk has finished being sent, but due to programmer error, the sender is missing an ack (perhaps for the last slice) and gets stuck in a state where it keeps resending this slice over and over and never getting an ack in return.

I've implemented this chunk system from scratch probably at least 5 times in the past 10 years and each time I find new and exciting ways to hang the sender. My strategy for developing and testing the chunk system is to first code it so that it works, and then setup a test harness that randomly sends chunks across of random sizes under large amounts of packet loss, out of order packets and duplicates. This tends to flush out any hangs. Not once have I managed to implement the chunk system without having at least one hang in it. So reader, don't get cocky if you are planning on implementing this from scratch. Please have a soak test. You can thank me later.

The first hang I usually encounter is caused by not sending acks back for redundant receives of the same slice. It goes a bit like this: "oh, this slice has already been received? discard the slice packet" and forgetting to set the dirty flag for an ack. This breaks the sender because without an ack, how will the sender ever realize the slice it repeatedly sending has been received? You'll only see this hang if you get particularly unlucky with packet loss, eg. it's necessary to lose the very last ack packet sent in response to the last slice in order for this condition to occur.

The next hang happens because the receiver knows the chunk send has completed before the sender does, so it transitions into the state 'readyToRead' which drops incoming packets. Again, in this state, even though the receiver thinks the chunk has been fully received, the sender doesn't yet know this, so it's necessary to keep setting that dirty flag for acks even once the chunk has been received, so the sender has acks sent back to it that lets it know the chunk is received.

The final hang that usually trips me up is on the transition after reading the chunk, where 'readyToRead' is set back to false and the chunk id is incremented. eg: chunk 0 finishes receiving, the user reads that chunk off and chunk id is incremented 1 so we are ready to start receiving slices for that chunk (we discard any slices with a different chunk id to the current slice we are receiving). Again, here the sender is a bit

behind and because of packet loss, may not get the first acks sent through. In this case, it is necessary to watch for slice packets and if we are in a state where we are not receiving chunk n yet, and a slice comes in for chunk n-1, then we must set a special dirty flag where we send out an ack for all slices in chunk n-1, otherwise the sender won't realize the chunk is received and will hang.

As you can see, acks are subtle and it's a strange process of induction where slices received cause more acks to be sent back in response that keeps this system moving forward and working. If you ever break this chain of slice -> ack response then the system will hang. I encourage you to take a close look at the source code for this article for more details.

Conclusion

It's a simple system in concept, but the implementation is certainly not trivial. In my opinion, it's actually a lot of fun learning experience to implement your own basic reliability system and chunk sender to get large blocks of data over UDP quickly and reliably. There's just so much to learn when you implement a system like this from scratch.

I hope you enjoy the design for this system and try your own hand at implementing it from scratch. It's a great learning experience. Alternatively, or perhaps in supplement, I encourage you to support me on patreon and in return you'll get the example source code for this article (and all other articles in this series), as well as the example source code for my GDC 2015 talk on Networked Physics.

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 3.0 licence. **Thanks for your support!**

Original URL:
http://gafferongames.com/building-a-game-network-protocol/sending-large-blocks-of-data/