gafferongames.com

Reliability, Ordering and Congestion Avoidance over UDP

13 min read • original

Introduction

Hi, I'm Glenn Fiedler and welcome to the fourth article in my series Networking for Game Programmers

In the previous article, we added our own concept of virtual connection on top of UDP.

Now we're going to add reliability, ordering and congestion avoidance to our virtual UDP connection.

This is by far the most complicated aspect of low-level game networking so this is going to be a pretty intense article, so strap in and lets go!

The Problem with TCP

Those of you familiar with TCP know that it already has its own concept of connection, reliability-ordering and congestion avoidance, so why are we rewriting our own mini version of TCP on top of UDP?

The issue is that multiplayer action games rely on a steady stream of packets sent at rates of 10 to 30 packets per second, and for the most part, the data contained is these packets is so time sensitive that only the most recent data is useful. This includes data such as player inputs, the position orientation and velocity of each player character, and the state of physics objects in the world.

The problem with TCP is that it abstracts data delivery as a reliable ordered stream. Because of this, if a packet is lost, TCP has to stop and wait for that packet to be resent. This interrupts the steady stream of packets because more recent packets must wait in a queue until the resent packet arrives, so the packets may be presented in order.

What we need is a different type of reliability. Instead of having all data treated as a reliable ordered stream, we want to send packets at a steady rate and get notified when packets are received by the other computer. This allows time sensitive data to get through without waiting for resent packets, while letting us make our own decision about how to handle packet loss at the application level.

It is not possible to implement a reliability system with these properties using TCP, so we have no choice but to roll our own on top of UDP.

Unfortunately, reliability is not the only thing that we must rewrite. This is because TCP also provides congestion avoidance so that it can dynamically scale the rate of data sent to suit the properties of the connection. For example TCP sends less data over a 28.8k modem than it would over T1 line, and it is able to do so without knowing in advance what sort of connection it is!

Sequence Numbers

Now back to reliability!

The goal of our reliability system is simple: we want to know which packets arrive at the other side of the connection.

First we need a way to identify packets.

What if we had added the concept of a “packet id”? Lets make it an integer value. We could start this at zero then with each packet we send, increase the number by one. The first packet we send would be packet 0, and the 100th packet sent is packet 99.

This is actually quite a common technique. It’s even used in TCP! These packet ids are called sequence numbers. While we’re not going to implement reliability exactly as TCP does, it makes sense to use the same terminology, so we’ll call them sequence numbers from now on.

Since UDP does not guarantee the order of packets, the 100th packet received is not necessarily the 100th packet sent. It follows that we need to insert the sequence number somewhere in the packet, so that the computer at the other side of the connection knows which packet it is.

We already have a simple packet header for the virtual connection from the previous article, so we’ll just add the sequence number in the header like this:

```
[uint protocol id]
[uint sequence]
(packet data...)
```

Now when the other computer receives a packet it knows its sequence number according to the computer that sent it.

Acks

Now that we can identify packets using sequence numbers, the next step is to let the other side of the connection know which packets we receive.

Logically this is quite simple, we just need to take note of the sequence number of each packet we receive, and send those sequence numbers back to the computer that sent them.

Because we are sending packets continuously between both machines, we can just add the ack to the packet header, just like we did with the sequence number:

```
[uint protocol id]
[uint sequence]
[uint ack]
(packet data...)
```

Our general approach is as follows:

- Each time we send a packet we increase the *local sequence number*
- When we receieve a packet, we check the sequence number of the packet against the sequence number of the most recently received packet, called the *remote sequence number*. If the packet is more recent, we update the remote sequence to be equal to the sequence number of the packet.
- When we compose packet headers, the local sequence becomes the sequence number of the packet, and the remote sequence becomes the ack.

This simple ack system works provided that one packet comes in for each packet we send out.

But what if packets clump up such that two packets arrive before we send a packet? We only have space for one ack per-packet, so what do we do?

Now consider the case where one side of the connection is sending packets at a faster rate. If the client sends 30 packets per-second, and the server only sends 10 packets per-second, we need *at least* 3 acks included in each packet sent from the server.

Lets make it even more complex! What if the packet containing the ack is lost? The computer that sent the packet would think the packet got lost but it was actually received!

It seems like we need to make our reliability system... *more reliable!*

Reliable Acks

Here is where we diverge from TCP.

What TCP does is maintain a sliding window where the ack sent is sequence number of the next packet it expects to receive, in order. If TCP does not receive an ack for a given packet, it stops and resends a packet with that sequence number again. This is exactly the behavior we want to avoid!

So in our reliability system, we never resend a packet with a given sequence number. We sequence n exactly once, then we send n+1, n+2 and so on. We never stop and resend packet n if it was lost, we leave it up to the application to compose a new packet containing the data that was lost, if necessary, and this packet gets sent with a new sequence number.

Because we're doing things differently to TCP, its now possible to have *holes* in the set of packets we ack, so it is no longer sufficient to just state the sequence number of the most recent packet we have received.

We need to include multiple acks per-packet.

How many acks do we need?

As mentioned previously we have the case where one side of the connection sends packets faster than the other. Lets assume that the worst case is one side sending no less than 10 packets per-second, while the other sends no more than 30. In this case, the average number of acks we'll need per-packet is 3, but if packets clump up a bit, we would need more. Lets say 6-10 worst case.

What about acks that don't get through because the packet containing the ack is lost?

To solve this, we're going to use a classic networking strategy of using redundancy to defeat packet loss!

Lets include 33 acks per-packet, and this isn't just going to be up to 33, but *always* 33. So for any given ack we redundantly send it up to 32 additional times, just in case one packet with the ack doesn't get through!

But how can we possibly fit 33 acks in a packet? At 4 bytes per-ack thats 132 bytes!

The trick is to represent the 32 previous acks before "ack" using a bitfield, like this:

```
[uint protocol id]
[uint sequence]
[uint ack]
[uint ack bitfield]
(packet data...)
```

We define "ack bitfield" such that each bit corresponds to acks of the 32 sequence numbers before "ack". So lets say "ack" is 100. If the first bit of "ack bitfield" is set, then the packet also includes an ack for packet 99. If the second bit is set, then packet 98 is acked. This goes all the way down to the 32nd bit for packet 68.

Our adjusted algorithm looks like this:

- Each time we send a packet we increase the *local sequence number*
- When we receive a packet, we check the sequence number of the packet against the *remote sequence number*. If the packet sequence is more recent, we update the remote sequence number.

- When we compose packet headers, the local sequence becomes the sequence number of the packet, and the remote sequence becomes the ack. The ack bitfield is calculated by looking into a queue of up to 33 packets, containing sequence numbers in the range [remote sequence – 32, remote sequence]. We set bit n (in [1,32]) in ack bits to 1 if the sequence number remote sequence – n is in the received queue.
- Additionally, when a packet is received, ack bitfield is scanned and if bit n is set, then we acknowledge sequence number packet sequence – n, if it has not been acked already.

With this improved algorithm, you would have to lose 100% of packets for more than a second to stop an ack getting through. And of course, it easily handles different send rates and clumped up packet receives.

Detecting Lost Packets

Now that we know what packets are received by the other side of the connection, how do we detect packet loss?

The trick here is to flip it around and say that if you don't get an ack for a packet within a certain amount of time, then we consider that packet lost.

Given that we are sending at no more than 30 packets per second, and we are redundantly sending acks roughly 30 times, if you don't get an ack for a packet within one second, it is *very* likely that the packet was lost.

So we are playing a bit of a trick here, while we can know 100% for sure which packets get through, but we can only be *reasonably* certain of the set of packets that didn't arrive.

The implication of this is that any data which you resend using this reliability technique needs to have its own message id so that if you receive it multiple times, you can discard it. This can be done at the application level.

Handling Sequence Number Wrap-Around

No discussion of sequence numbers and acks would be complete without coverage of sequence number wrap around!

Sequence numbers and acks are 32 bit unsigned integers, so they can represent numbers in the range [0,4294967295]. Thats a very high number! So high that if you sent 30 packets per-second, it would take over four and a half years for the sequence number to wrap back around to zero.

But perhaps you want to save some bandwidth so you shorten your sequence numbers and acks to 16 bit integers. You save 4 bytes per-packet, but now they wrap around in only half an hour!

So how do we handle this wrap around case?

The trick is to realize that if the current sequence number is already very high, and the next sequence number that comes in is very low, then you must have wrapped around. So even though the new sequence number is *numerically* lower than the current sequence value, it actually represents a more recent packet.

For example, lets say we encoded sequence numbers in one byte (not recommended btw. :)), then they would wrap around after 255 like this:

```
... 252, 253, 254, 255, 0, 1, 2, 3, ...
```

To handle this case we need a new function that is aware of the fact that sequence numbers wrap around to zero after 255, so that 0, 1, 2, 3 are considered more recent than 255. Otherwise, our reliability system stops working after you receive packet 255.

Here it is:

```
bool sequence_more_recent( unsigned int s1,
                           unsigned int s2,
                           unsigned int max )
{
    return
        ( s1 > s2 ) &&
        ( s1 - s2 <= max/2 )
           ||
        ( s2 > s1 ) &&
        ( s2 - s1  > max/2 );
}
```

This function works by comparing the two numbers *and* their difference. If their difference is less than 1/2 the maximum sequence number value, then they must be close together – so we just check if one is greater than the other, as usual. However, if they are far apart, their difference will be greater than 1/2 the max sequence, then we paradoxically consider the sequence number more recent if it is *less* than the current sequence number.

This last bit is what handles the wrap around of sequence numbers transparently, so 0,1,2 are considered more recent than 255.

So simple and elegant!

Make sure you include this in any sequence number processing you do!

Congestion Avoidance

While we have solved reliability, there is still the question of congestion avoidance. TCP provides congestion avoidance as part of the packet when you get TCP reliability, but UDP has no congestion avoidance whatsoever!

If we just send packets without some sort of flow control, we risk flooding the connection and inducing severe latency (2 seconds plus!) as routers between us and the other computer become congested and

buffer up packets. This happens because routers try *very hard* to deliver all the packets we send, and therefore tend to buffer up packets in a queue before they consider dropping them.

While it would be nice if we could tell the routers that our packets are time sensitive and should be dropped instead of buffered if the router is overloaded, we can't really do this without rewriting the software for all routers in the world!

So instead, we need to focus on what we can actually do which is to avoid flooding the connection in the first place.

The way to do this is to implement our own basic congestion avoidance algorithm. And I stress basic! Just like reliability, we have no hope of coming up with something as general and robust as TCP's implementation on the first try, so lets keep it as simple as possible.

Measuring Round Trip Time

Since the whole point of congestion avoidance is to avoid flooding the connection and increasing round trip time (RTT), it makes sense that the most important metric as to whether or not we are flooding our connection is the RTT itself.

We need a way to measure the RTT of our connection.

Here is the basic technique:

- For each packet we send, we add an entry to a queue containing the sequence number of the packet and the time it was sent.
- Each time we receive an ack, we look up this entry and note the difference in local time between the time we receive the ack, and the time we sent the packet. This is the RTT time for that packet.
- Because the arrival of packets varies with network jitter, we need to smooth this value to provide something meaningful, so each time we obtain a new RTT we move a percentage of the distance between our current RTT and the packet RTT. 10% seems to work well for

me in practice. This is called an exponentially smoothed moving average, and it has the effect of smoothing out noise in the RTT with a low pass filter.
- To ensure that the sent queue doesn't grow forever, we discard packets once they have exceeded some maximum expected RTT. As discussed in the previous section on reliability, it is exceptionally likely that any packet not acked within a second was lost, so one second is a good value for this maximum RTT.

Now that we have RTT, we can use it as a metric to drive our congestion avoidance. If RTT gets too large, we send data less frequently, if its within acceptable ranges, we can try sending data more frequently.

Simple Binary Congestion Avoidance

As discussed before, lets not get greedy, we'll implement a very basic congestion avoidance. This congestion avoidance has two modes. Good and bad. I call it simple binary congestion avoidance.

Lets assume you send packets of a certain size, say 256 bytes. You would like to send these packets 30 times a second, but if conditions are bad, you can drop down to 10 times a second.

So 256 byte packets 30 times a second is around 64kbits/sec, and 10 times a second is roughly 20kbit/sec. There isn't a broadband network connection in the world that can't handle at least 20kbit/sec, so we'll move forward with this assumption. Unlike TCP which is entirely general for any device with any amount of send/recv bandwidth, we're going to assume a minimum supported bandwidth for devices involved in our connections.

So the basic idea is this. When network conditions are "good" we send 30 packets per-second, and when network conditions are "bad" we drop to 10 packets per-second.

Of course, you can define “good” and “bad” however you like, but I’ve gotten good results considering only RTT. For example if RTT exceeds some threshold (say 250ms) then you know you are probably flooding the connection. Of course, this assumes that nobody would normally exceed 250ms under non-flooding conditions, which is reasonable given our broadband requirement.

How do you switch between good and bad? The algorithm I like to use operates as follows:

- If you are currently in good mode, and conditions become bad, immediately drop to bad mode
- If you are in bad mode, and conditions have been good for a specific length of time ‘t’, then return to good mode
- To avoid rapid toggling between good and bad mode, if you drop from good mode to bad in under 10 seconds, double the amount of time ‘t’ before bad mode goes back to good. Clamp this at some maximum, say 60 seconds.
- To avoid punishing good connections when they have short periods of bad behavior, for each 10 seconds the connection is in good mode, halve the time ‘t’ before bad mode goes back to good. Clamp this at some minimum like 1 second.

With this algorithm you will rapidly respond to bad conditions and drop your send rate to 10 packets per-second, avoiding flooding of the connection. You’ll also *conservatively* try out good mode, and persist sending packets at a higher rate of 30 packets per-second, while network conditions are good.

Of course, you can implement much more sophisticated algorithms. Packet loss % can be taken into account as a metric, even the amount of network jitter (time variance in packet acks), not just RTT.

You can also get much more *greedy* with congestion avoidance, and attempt to discover when you can send data at a much higher bandwidth (eg. LAN), but you have to be very careful! With increased greediness comes more risk that you’ll flood the connection!

Conclusion

Our new reliability system lets us send a steady stream of packets and notifies us which packets are received. From this we can infer lost packets, and resend data that didn't get through if necessary.

On top of this we have a simple congestion avoidance system that alternates between sending packets 10 times a second and 30 times a second, depending on network conditions, so we don't flood the connection.

There are lots of implementation details too specific to mention in this article, so make sure you check out the example source code to see how it is all implemented.

Thats it for reliability, ordering and congestion avoidance, probably the most complicated aspect of low-level networking.

If you enjoyed this article please consider making a small donation.
Donations encourage me to write more articles!

Original URL:
http://gafferongames.com/networking-for-game-programmers/reliability-and-flow-control/