Networking: How a Shooter Shoots

By Armin Ronacher Published December 22, 2011

I'm going to lean out of the window here and share some knowledge I gathered about a very complex topic I had no idea about a few months ago. Why am I doing that? Because I found it fascinating to read about it and maybe it gives someone else a reason to further dive into that kind of stuff. And despite all my searching I did not find a whole lot about this whole topic that would reference more dynamic games like Battlefield. I don't claim competence and what's even worse is that a lot of that stuff seems to be surprisingly hard to dig up.

First a little bit of context: I have to admit that I did sink an awful lot of time into the Battlefield series after Bad Company 2 and now with Battlefield 3 and whenever I find myself spending too much time on "useless" things like that I want to find a way to at least learn something from it that I might be able to use at one point. The nice part about DICE is that the developers are quite active on forums, twitter and reddit and you get a bit of information about how stuff is being developed. Also the game is popular enough that many people poke around in the executable to see how it operates for better or worse.

In case of Battlefield I was curious how the networking side of things worked. Mainly because the game was doing one thing different than anything else I played before: each and every bullet fired has a velocity and is affected by gravity. While it's true that projectiles are not a new concept, very few games attempt to lag compensate them. In fact I did not find any besides Bad Company 2 and Battlefield 3 but that might also just have been my lack of knowledge in that area.

Damn Physics

Problem number one is in general that information can only travel so fast. The speed of light is the theoretical limit, but practically it's less than that since we're using fiber and copper for data transmission. On top of that you have network switching equipment and more that adds latency. So there are three possible solutions: solution one is that your game is only played in local area networks where the latency is negligible low, solution two is ignoring the issue altogether and let the players compensate for the "lag" themselves, solution three would be to apply clever tricks to compensate for the latency.

While the network latency improved over time we're now getting close the physical limits and the problem is not gone. In fact we're not interacting with people from across the globe whereas previously communities were less distributed.

So what games are doing is tricking the player into believing that there is no lag. For many games you won't notice a latency of 150ms all that much from the general gameplay. You will notice that with raising latency your general performance declines but unless you are experienced you probably could not attribute it to the latency directly.

Basic Operation

A modern networked shooter usually consists of two entities. A host and a client. The host simulates the game at a specific rate (for instance 20 ticks per second) and will inform all the clients about changes on the server by sending out updates of the game world at a fixed update rate. The clients get that information and draw what's happening on their screen.

Now if the server only updates 20 times a second a naive solution would also mean that you're locked to 20Hz. Since you want more FPS, the client applies interpolation. Assuming the server ticks at 20Hz and for each tick on the server the client is sent a package, it would need to render three frames for each packet. The thing with interpolation is that it can only generate you data between points. So assuming 60Hz, you have ~16ms per frame, means that if interpolation happens only between one server tick you will be additional 48ms in the past for interpolation to work.

Alternatively one could theoretically extrapolate from the known points into the future but this is error prone and a lot more complex than a simple linear interpolation.

With that we have the foundation for simulating other player's replicated movements. But what about the local player? If the local player would move, then send his movements to the server and wait for his movement commands to come back there would be a high lag involved in that. The solution for that is Client-side prediction. In computer games the term Dead Reckoning often refers to the same concept. In a static world where nothing but the players changes, it's safe to say that the client could just ignore the server for its own movements and just simulate away the moment the movement buttons are pressed.

In case the simulation on the client and server diverge you will get artifacts such as a bit of teleporting of your player. When does the simulation diverge on server and client? Usually when the player bumps into something on the server which he did not predict on the client. For instance running into another player or standing on a vehicle that is not under the player's control.

A Time Machine

The fundamentals from the section above work good enough for a game like Quake or Counterstrike where the only dynamic elements in the game are players and rockets. If the client predicts incorrect stuff, the server will still send updates about the player's position. If the local machine sees that the server data and the local prediction is too far apart it has to correct.

Since however the data provided by the server will be "from the past" from the point or view of the current simulation, the game has to backtrack. It usually does that by keeping the last few network messages in a local buffer and finds the one that corresponds to the updated server message and rebases the simulation on top of the now corrected state.

Did I Hit Something?

Now the movements are sorted out, but not yet the shooting. In Quakeworld for instance the movement was nicely compensated for lag thanks to the prediction, but one would still have to lead the target to compensate for the latency in order to hit the opponent. If the shooter lags for 100ms and looks at a player moving

at 100 units per second one would have to lead 10 units to land a hit when shooting. Annoying.

If you play a modern shooter that's no longer the case. So how does that work? Basically the same way. In order to detect if something was hit the client transmits the aim direct to the server and announces that it attempts to shoot. The server looks at the time difference, "walks back in time" for that amount of time and sees if the shooter would have hit something. If he did hit, the damage is applied.

This has two affects: first of all it only works for infinitely fast weapons (hitscan) because no bullet is actually created that would traverse time and space. Secondly it has the effect that if the shooter has a latency of 200 milliseconds the person getting shot might already have been moved behind cover. Thanks to the shooter playing further in the past he still ended up being dead because he was clearly visible from the position of the shooter.

You can't get rid of the lag, you can just shift it to other places. In this case modern games decide on giving a lag free experience to the shooter. The illusion of playing at the same time is then of course destroyed once something extraordinary happens (players from different "times" running together, a player getting shot behind cover etc.).

Real Bullets

The first problem that Battlefield here has is that it creates bullets. In Quake and other games you have around 16 players. Each time you try to see if someone was hit the server basically just reverts all player's positions for the client's latency and interpolation time and sees if something is hit. Even with a very bullet spongy weapon that does not cause too many problems, especially since you could cheat a little bit there and combine multiple bullets into one network packet. While it is true that there were games before that traded hitscan weapons for projectiles, they did not apply lag compensation for it and made the projectiles comparatively slow. This made playing over high ping connections much harder. Try playing Team Fortress 2 with the medic gun on a high latency server and see how well you can adapt :-)

However in Battlefield you are generating actual bullets and each bullet has a TTL of 1.5 seconds. For sniper rifle bullets that even increases to 5 seconds. Considering that most weapons are assault rifles or worse you are looking at 700 RPM rifles there where a considerable amount of bullets misses. Worse: the maps are so large that many bullets will travel up to their maximum TTL before they are removed. On top of all that you are dealing with up to 64 players and vehicles as well.

Those are a lot of dynamic objects that all need to be checked for each simulation step if they hit something. And each time a hit check is performed the game would have to move all players back in time according to the bullets own time frame which was recorded as the time frame of the person firing. Each simulation step and not just each time when someone shoots.

Destruction and Vehicles

Another big problem that Battlefield has created itself with the game design is the fact that environment is destructible. Buildings and vehicles are possible receivers of damage and if you have to do the time travel to see if something was

shot you not only have to move players back in time, you have to do the same for all vehicles on the battlefield. I don't think it is necessary to do that for the destruction since the destruction does not cause new obstacles for bullets. The rubble can largely be shot through and in case a new obstacle does appear as part of the destruction process it will most likely not be noticed in all the smoke that happens while destruction is taking place.

However one part of the destruction in Battlefield 3 is the ground destruction. You can blow holes into the ground which must have a devastating effect on any kind of prediction or interpolation that is happening. Considering that during beta tons of players reported about soldiers falling under the map it does not come as a surprise that the ground destruction was toned down a lot for the final version.

Scaling the Protocol

With 64 players on the battlefield, dozens of vehicles, tons of destruction, rockets, bullets, weapons lying around and god knows what - there is a lot of data to transmit. If you look at the amount of traffic that Battlefield 3 actually transmits you will notice that it's not a lot. I would assume that it does something similar to the Halo network model which is based on the Tribes one. Halo improved on that by adding various degrees of priorities to all networked objects. As such objects further away from the player's view frustum are updated much less often than objects near the player.

On my machine on a 64 player server the game transmits around 16KB/sec in both directions. That's not a lot.

Halo's Ragdoll Trick

There are a lot of physics objects on the battlefield and I don't think many of them are networked. For instance ragdolls mostly just take up bandwidth and for as long as the initial physics state is synchronized to the clients you could mostly ignore them. You can already shoot through dead bodies anyways so the biggest problem is solved. The problem there is that Battlefield allows you to revive fallen comrades.

This is one example where I saw some people abusing the revive mechanics with cheating. If it's true what I found on certain websites that shall go unnamed, the player's position after revive is controlled by the reviving person. That would make sense from the user experience because the person being revived does not know what happened to his body while he was dead anyways since the camera points to somewhere else. And from the view of the person reviving it would be awkward to see the person being revived teleporting to the position the server thinks it should be located.

But regardless of the fact if Battlefield networks ragdolls or not, it's a interesting observation how game design can affect networking or the other way round. It certainly worked well for Halo where dead bodies serve no purpose unlike Battlefield 3 where they can be revived.

Hit Detection in Battlefield 3

So Battlefield 3 has real bullets and those bullets appear properly compensated for lag. If you hit something, you hit it. What's even more interesting is that it's

close to pixel perfect. It's hard to say for sure since the guns do not have 100% accuracy, even for the first shot. But it seems pretty damn close.

The way this works in Battlefield 3 is that the client does the hit detection, not the server. It's like the UT99 mod ZeroPing in that regard. Considering the complexities behind all the guns in Battlefield 3 it makes a lot of sense. The advantages of client side hit detection are obvious: cheap, pixel perfect hit detection, automatic lag compensation for free. It's the perfect solution. It however comes with the downside of being easy to cheat.

According to Alan Kertz, DICE compensates for the security implications of client side hit detection by performing a basic plausibility check on the server if the reported hit is likely or not. I would assume they are doing something indeed considering that the "killed across the map" hack videos died out with the end of the beta. Since the main problem regarding hacking on PC gaming are aimbots and ESP hacks anyways I'm not so sure if there is a net loss in terms of security in practice at all there.

How do you verify the hit on the server? You could create a cone from the origin and make it expand to the distance. Since you have to account for gravity that might not be the perfect solution but probably close enough.

Writing your own Network Game

So what do you need to know in order to make your own network game? For playing around with that kind of stuff there are a couple of good hints available on the internet.

- Glenn Fiedler's game development articles on Networking for Game Programmers and Networked Game Physics are a very good start.
- Gamasutra has an article from 1997 about Dead Reckoning which is a good introduction to the topic.
- Fabien Sanglard's Quake engine code review covers the network and prediction parts. In general the whole Quake engine is a good place to start reading about networking.
- Valve's developer wiki is an amazing source of information. They have articles about Lag compensation, Prediction, Interpolation and Yahn Bernier's paper on game engine networking among other useful things.
- The Halo I shot you first talk at the GDC vault.
- Austin GDC 2008 talk about Robust Efficient Networking by Ben Garney.

This is all the stuff I have used so far. I am sure there are tons of books on the topic but I do not have any recommendations there.