FPS Network Solution

By lxh 2019.5.14

1. **传输协议决策**

**1、TCP与UDP对比**

|  |  |  |
| --- | --- | --- |
|  | TCP | UDP |
| 优点 | ①可靠性传输  ②保证数据包有序，无重复，有丢包处理，拥塞处理 | ①简单  ②传输效率高  ③延时影响低 |
| 缺点 | 可靠性传输+复杂的拥塞处理/重传机制，导致延时影响高，流量控制导致传输效率降低 | ①unorder：数据包无序  ②duplicate：数据包可能重复  ③dismiss：丢包后无任何处理 |

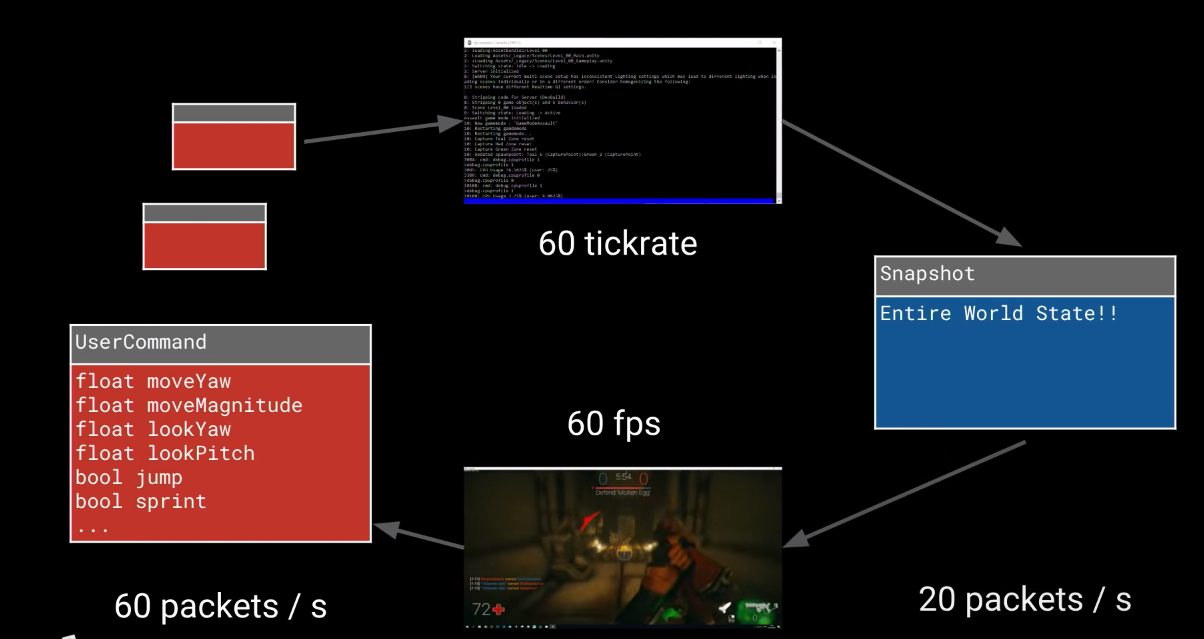
**2、FPS需求分析**

FPS玩家操作的及时反馈要求很高，换句话讲就是网络延时的影响要足够的低，因此最好采用UDP协议，通常会基于UDP在应用层实现一套RUDP(Reliable UDP协议)，后面会提到

1. **网络同步方案总结(Sample Game)**

**0、Tick Rate**(\*)

|  |  |
| --- | --- |
| Server | Client |
| Tick to send snapshot：20/s | Tick to receive/handle snapshot：20/s |
| Tick to receive command/logic update：60 /s | Tick to send command/render: 60/s |



**1、采用UDP需要解决的问题**

**<1>无序/重复**

UDP的数据段加一个32bit序列号(帧号)Seq Num(\*)

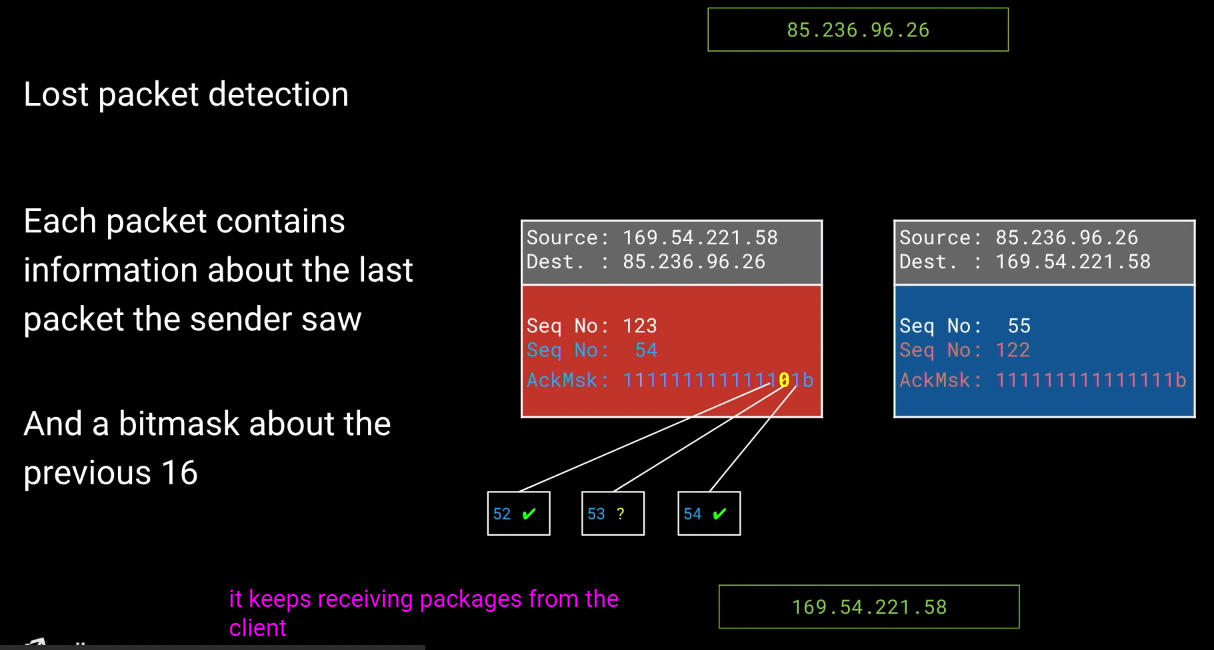


**<2>丢包**

UDP的数据段加入(\*)：

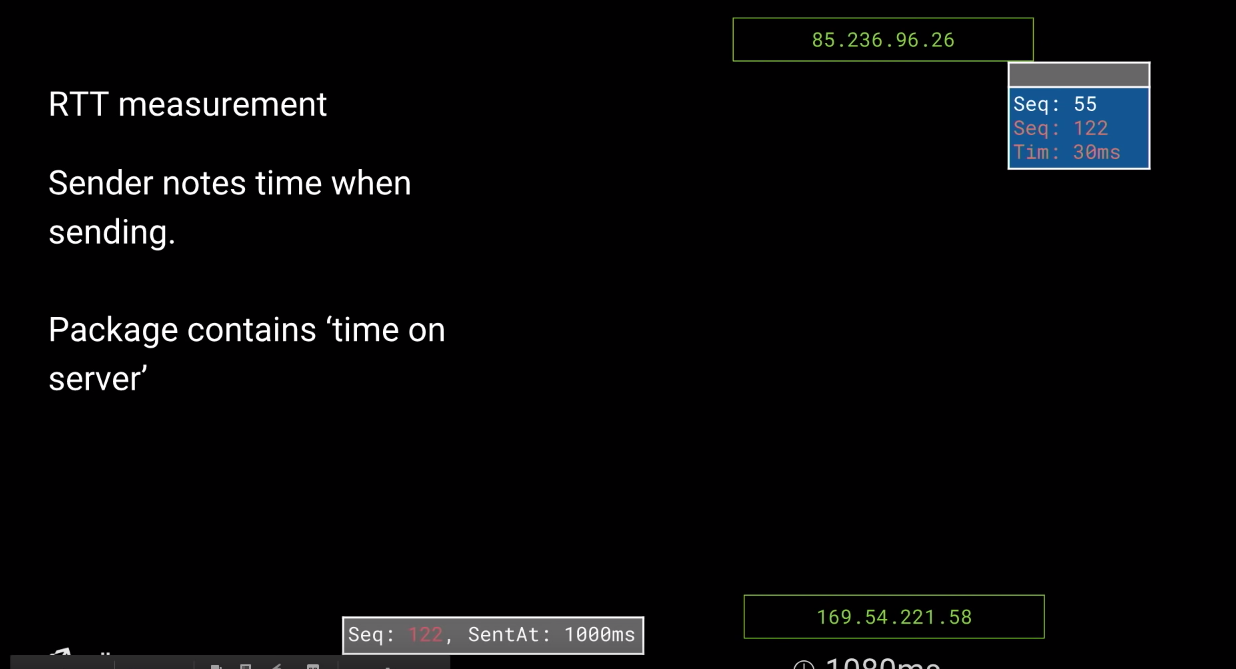
① ‘Last Ack Seq Num’: last received ack pack from the other side(32 bit)

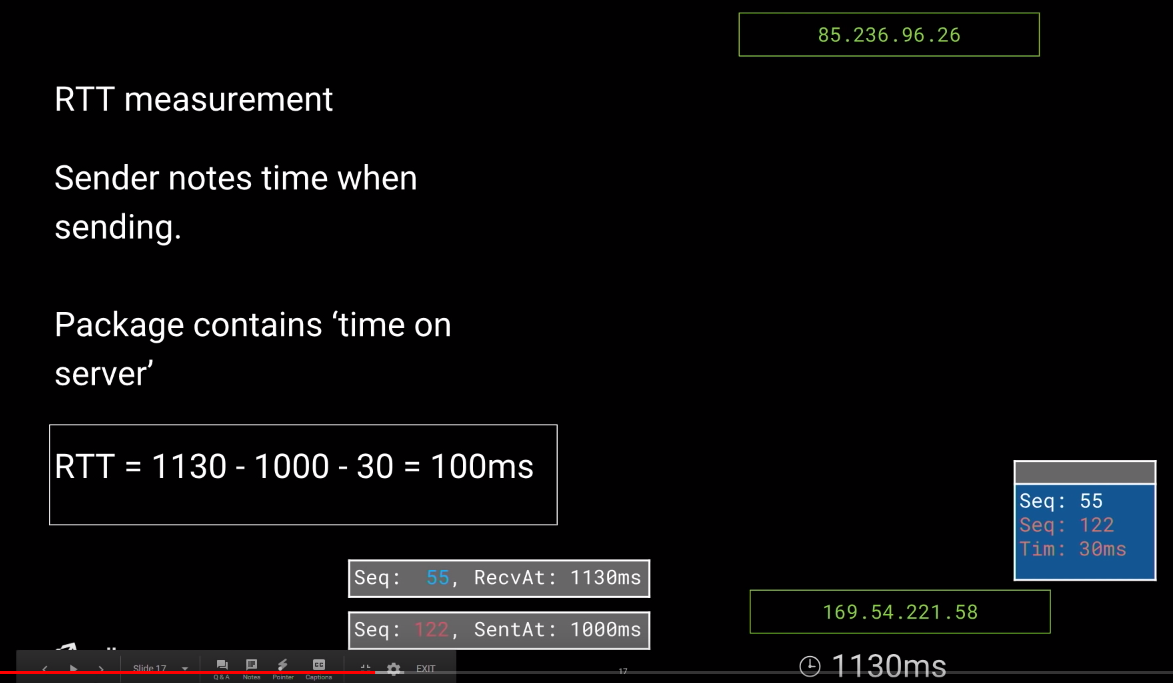
② AckMask(16 bit)：表示包括‘Last Rev Seq Num’以及其前15帧数据收到情况(1, received; 0, not receive yet)，如果服务器长时间(从代码中看似乎是丢包后立马重传)收到客户端某个Seq Num的帧数据未收到，视为丢包，需要重传



**<3>RTT处理**

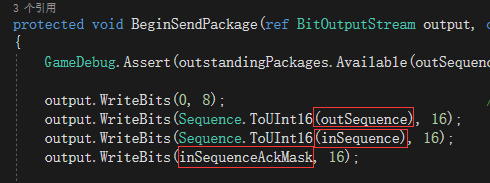


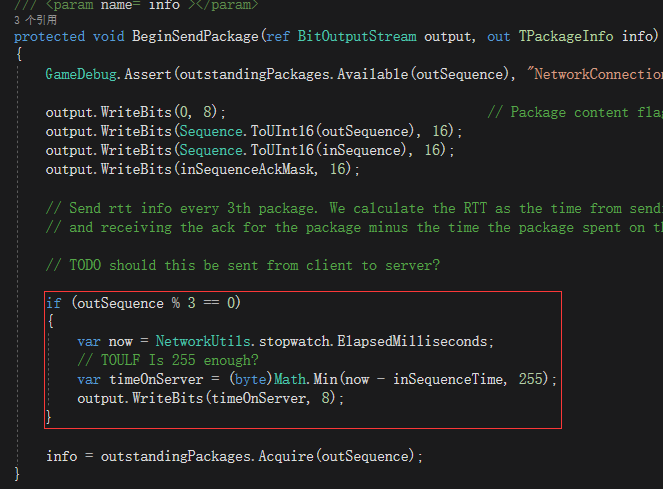




Sample game中客户端每3个渲染帧处理一次RRT校验(\*)

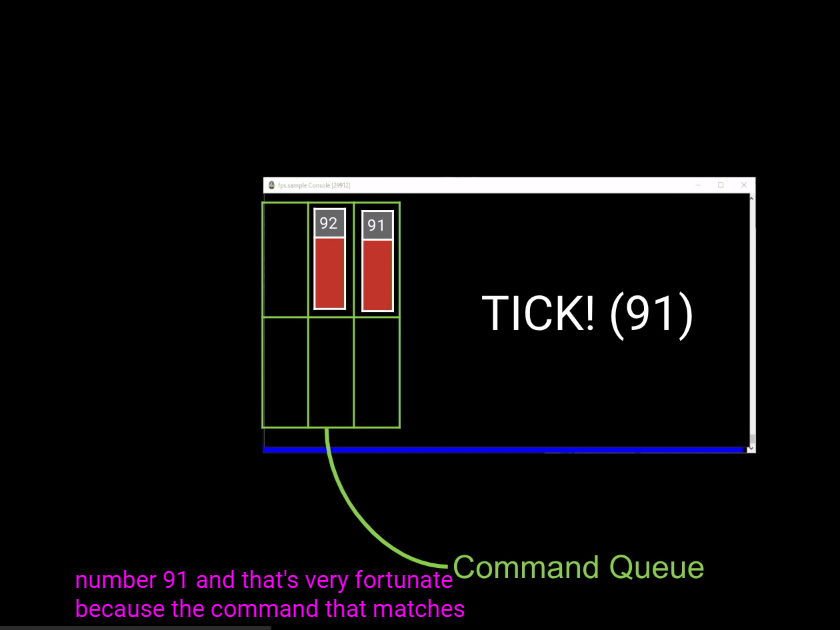
**<4>Code Review**



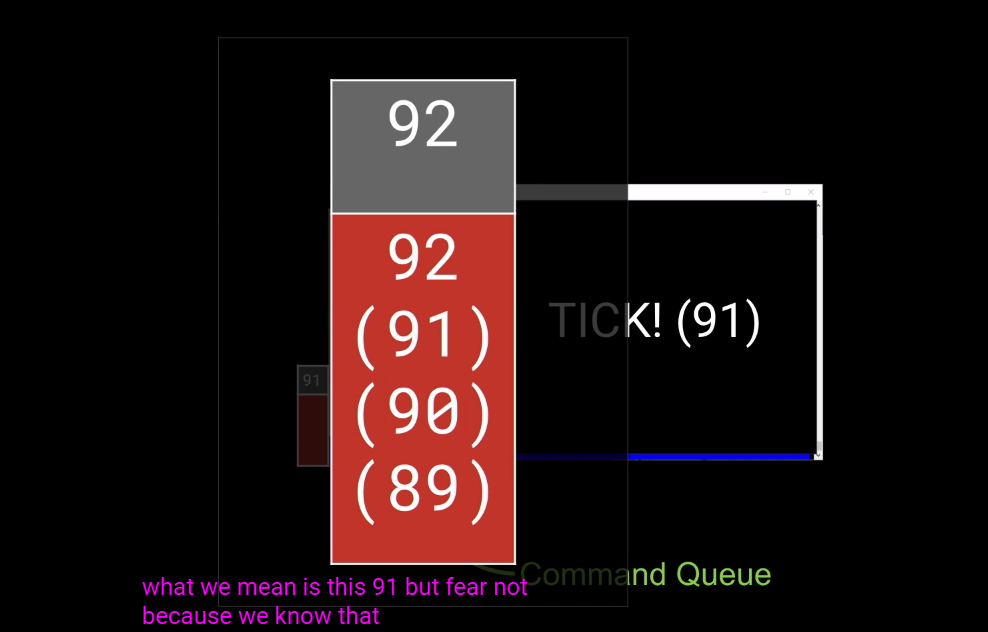


**2、少量丢包处理**(\*)

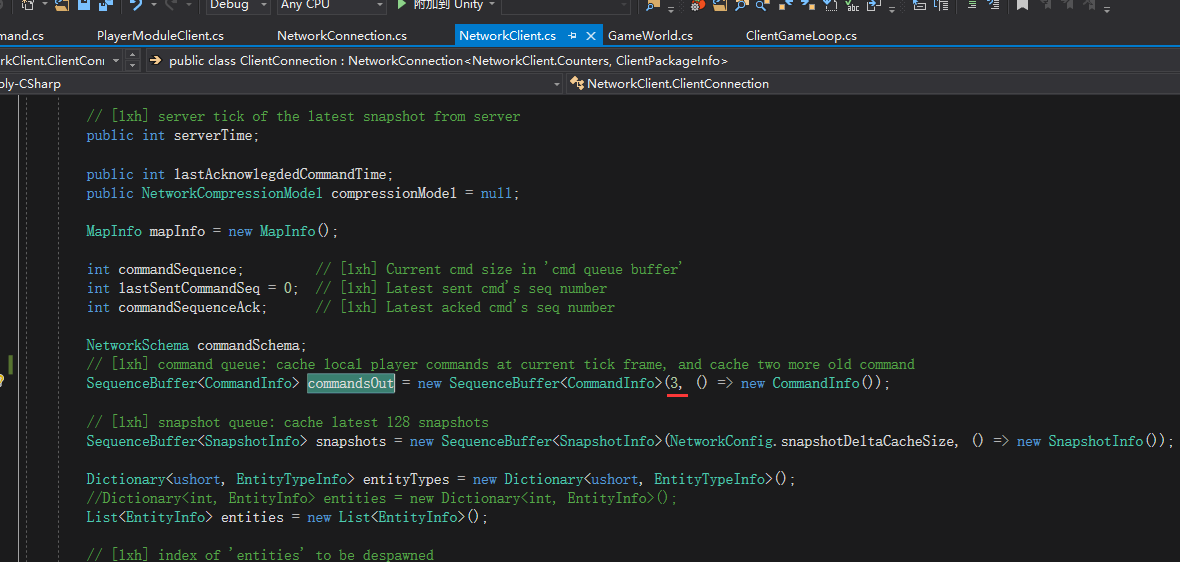
① Server保存一个Command Queue(保证有序的)(\*)

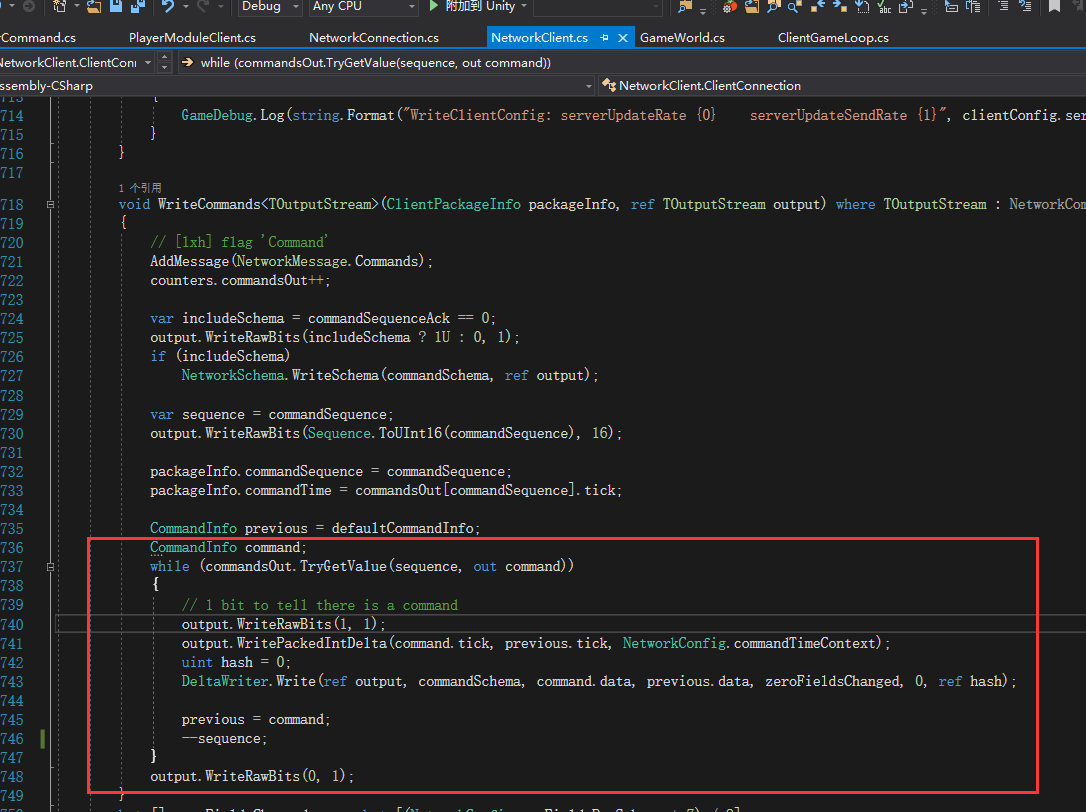


② 每一个操作数据包包括当前帧以及前3帧的操作，这样就算丢失了2~3个数据包也没关系，可以从已收到的帧数据包中恢复



Code View

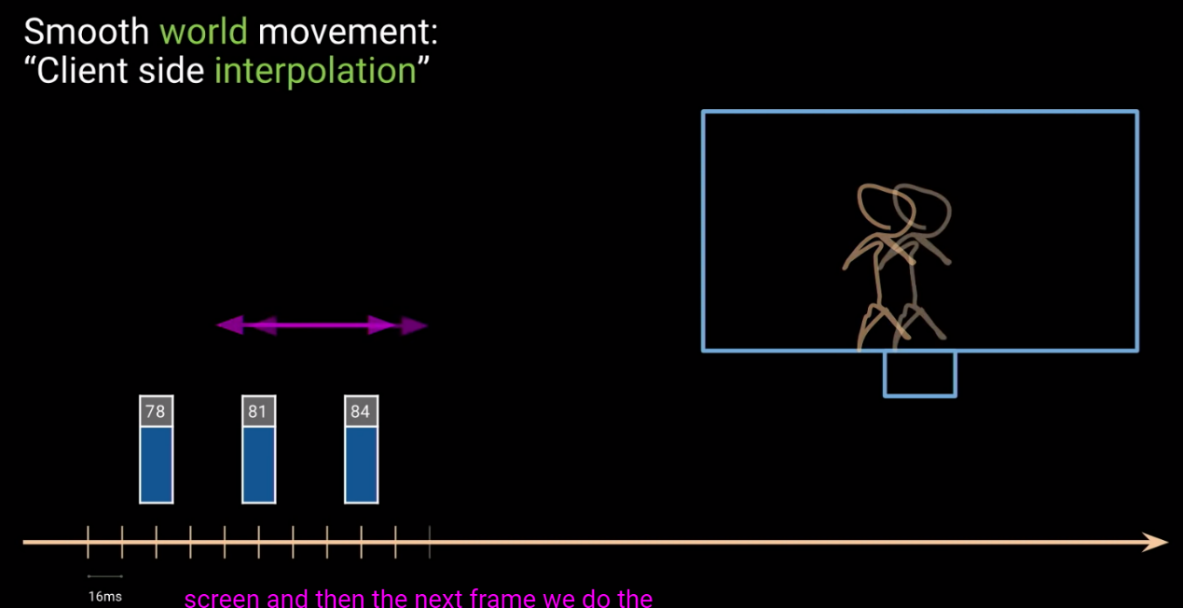




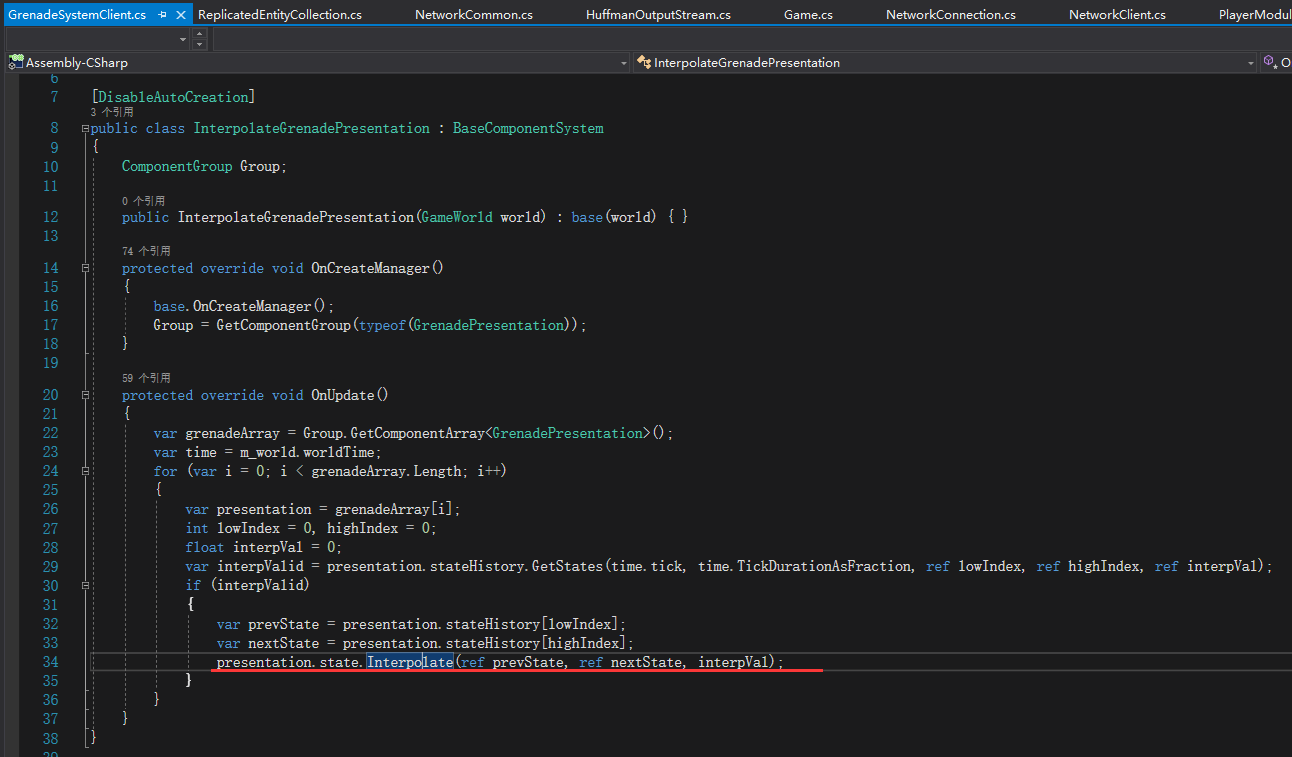
**3、客户端平滑表现**

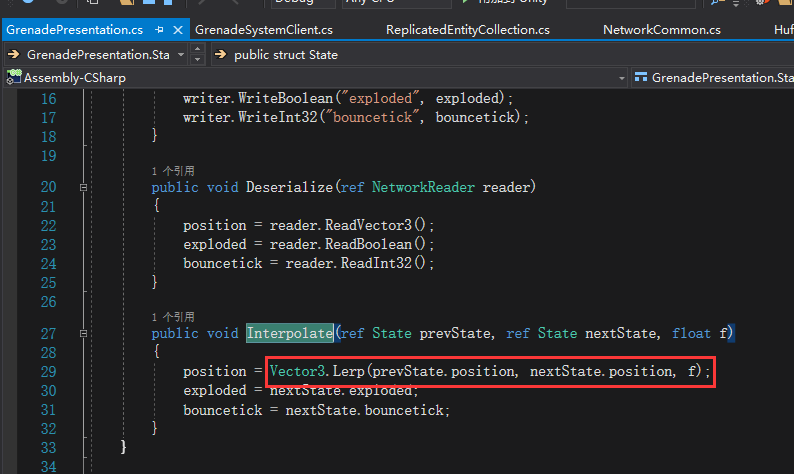
线性插值(以Move为例)(\*)

由于服务器发送的快照Snapshot是以20tick/s的速度，因此每两个Snapshot之间的存在一定数据量的差异，如果直接直接设置每个Snapshot下的坐标信息，客户端角色可能会有瞬移的不平滑表现，为了解决这个问题，在Snapshot(i)和Snapshot(i+1)之间做一个线性插值，以让客户端角色的移动更加平滑。



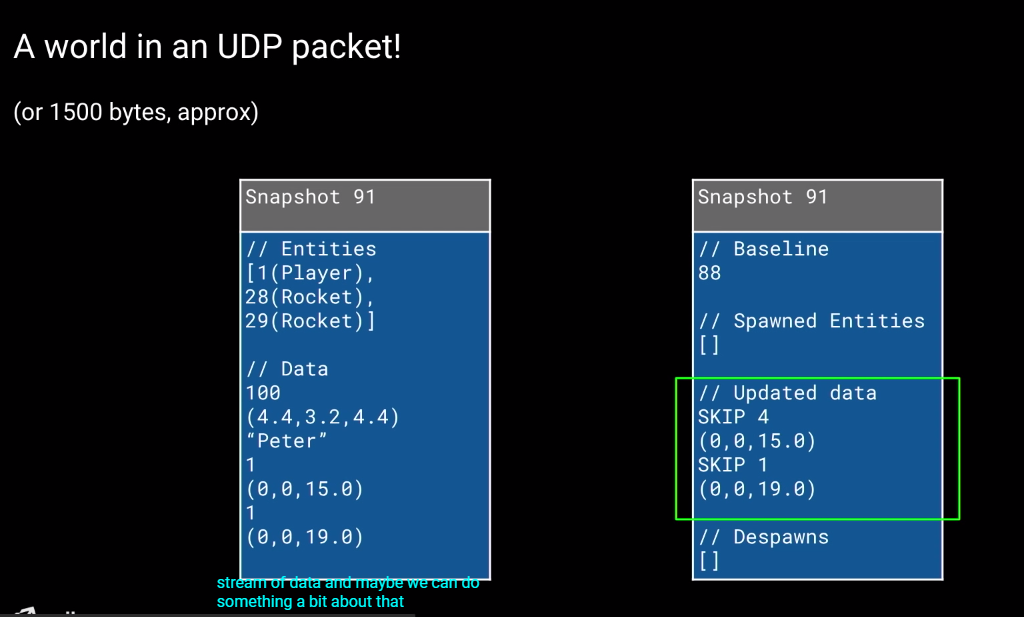
**Code Review**





**4、UDP包体压缩优化**(\*)

直接传输Snapshot，数据量会比较大，在受到MTU传输限制下，考虑采用’Delta Snapshot ‘，即只传输快照差值



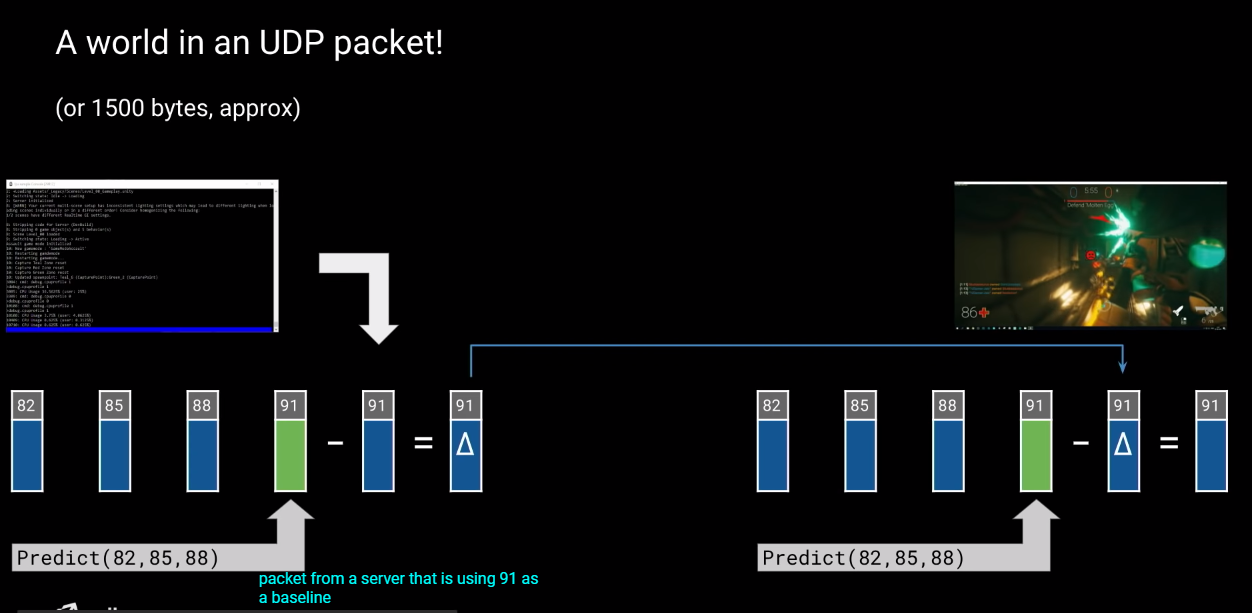
以3个基准线为例，假设当前客户端tick在88，且Client已经收到Server发送的Snapshot82、85和88，对于下一个Snapshot91(Delta Snapshot from 88):

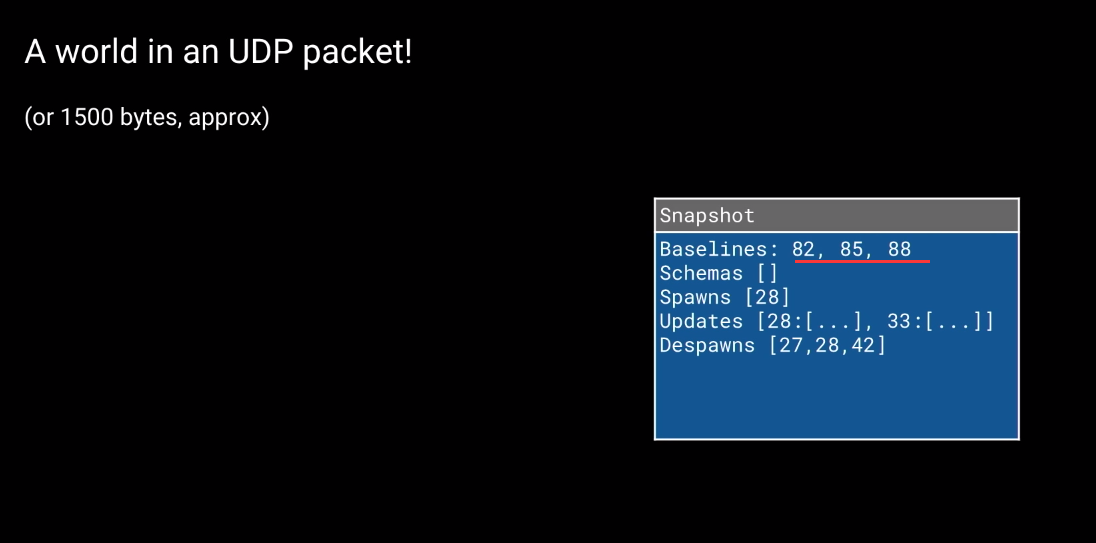
Client和Server采用同样的预测算法函数，基于现有的Snapshot82、85、88预测一个Snapshot91\_Predict，然后Server得到一个真正的Snapshot91\_Real，传输给Client一个delta Snapshot91:

Delta Snapshot91 = Snapshot91\_Predict - Snapshot91\_Real

客户端收到Delta Snapshot91，首先也会使用同样的预测算法，基于过去的3个Snapshot预测出Snapshot91\_Predict，然后通过简单的计算得出真正的Snapshot91：

Snapshot91\_Real = Snapshot91\_Predict - Delta Snapshot91





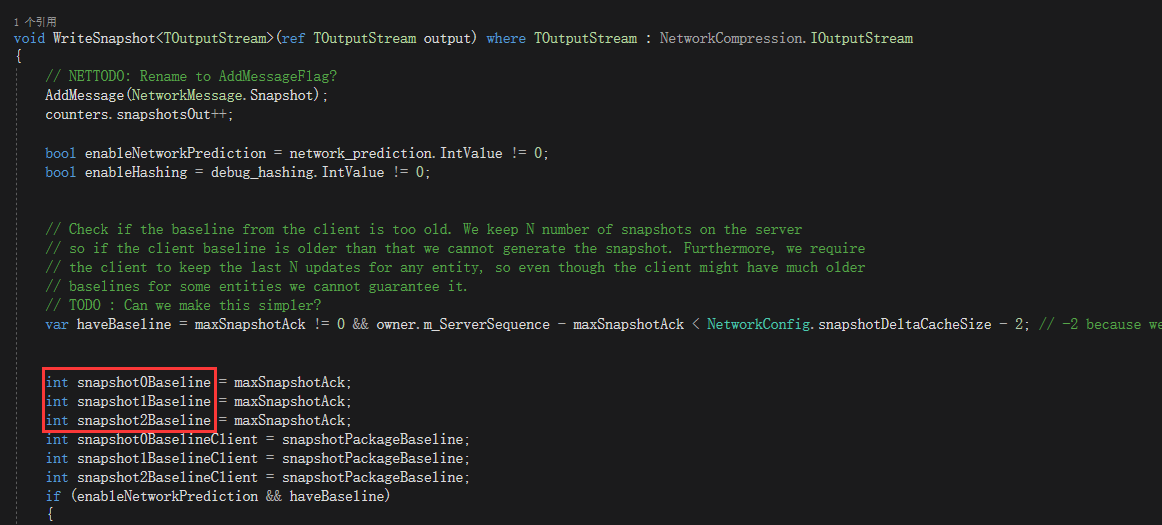
这样做的好处和理论依据在于：

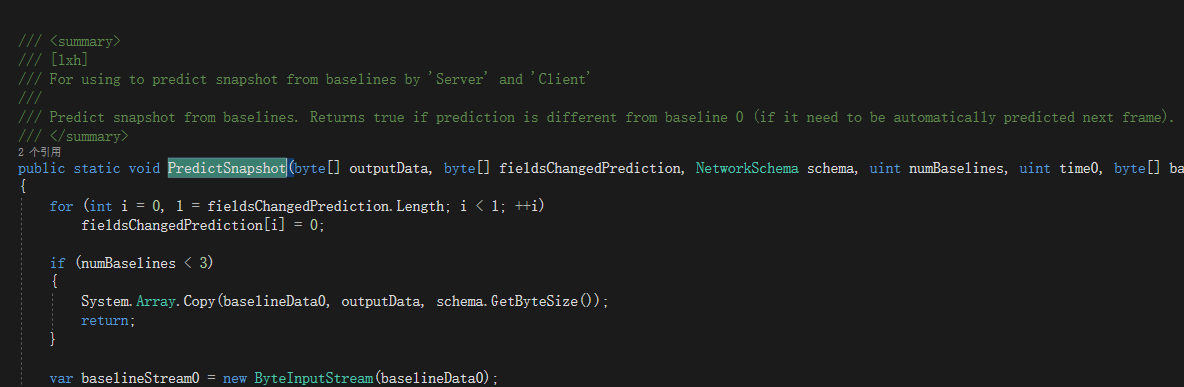
①大多数Entity的属性发生变化的频率都不高(eg, 墙、树等静态物体，Player的属性(血量、防御等))

②预测算法足够精准(具体预测细节还没仔细研究，可以后期优化时再考虑)

这样一来，Delta Snapshot会得到很多0，大大提高网络数据包体的压缩效率(btw, Sample采用的是Huffman)

**Code Review**

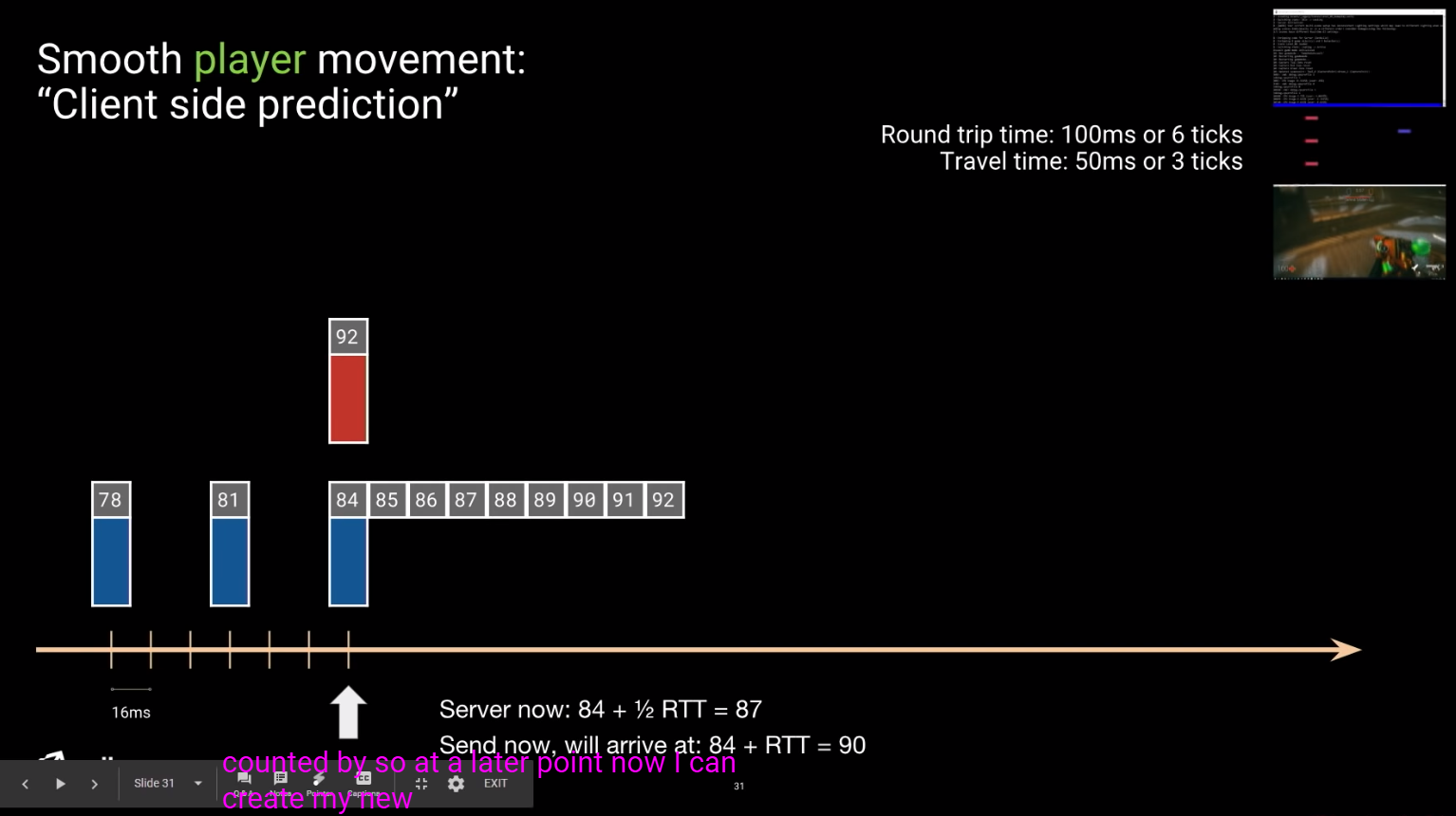


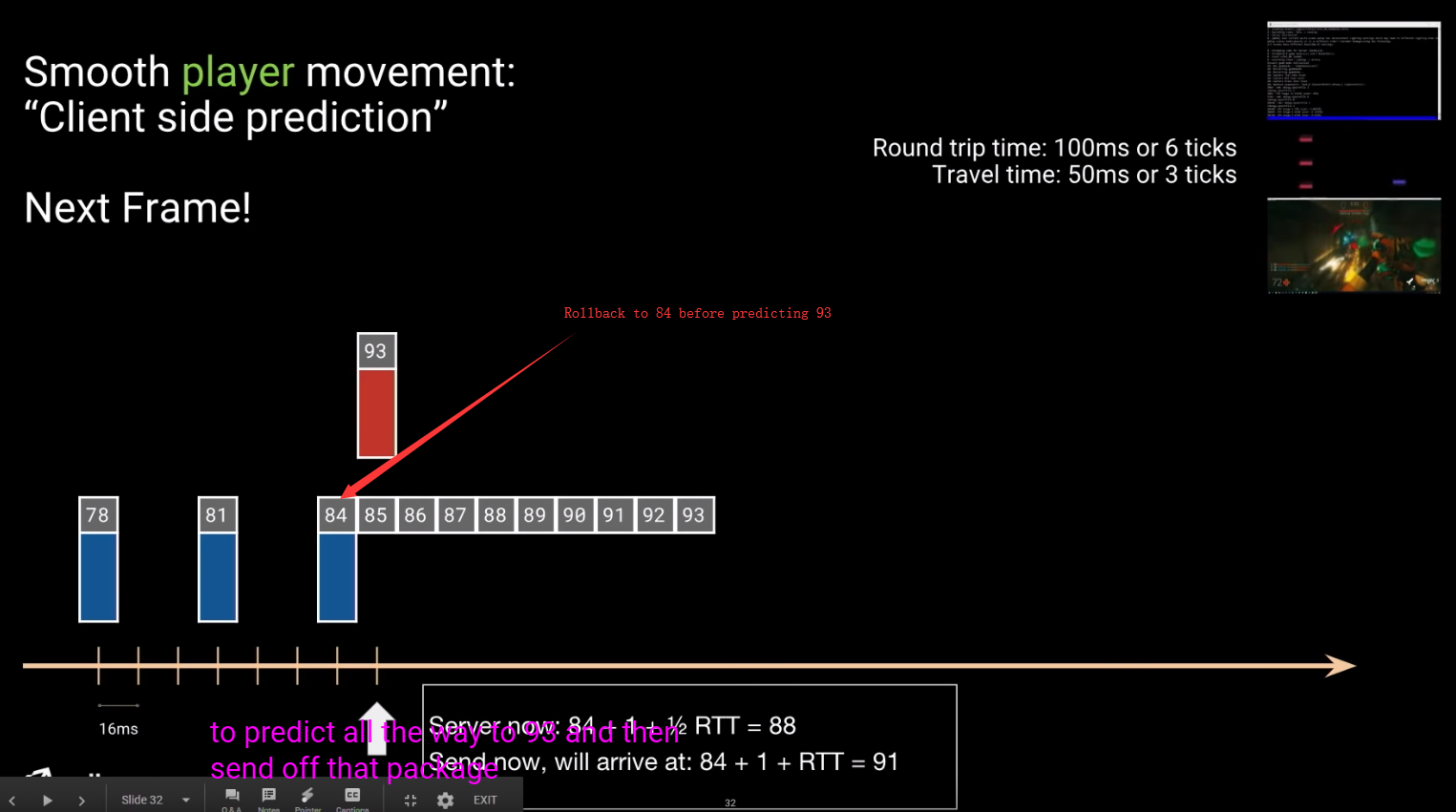


**5、Local Player Move操作及时反馈处理**(\*)

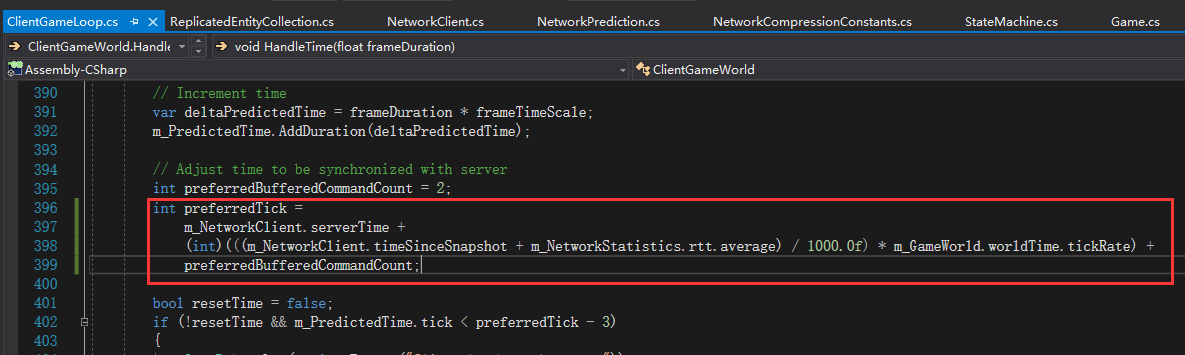
Local Player操作命令后，并发送给Server后，由于网络时延以及Snapshot发送帧率低于TickRate，导致Local Player不能及时收到下一个新的Snapshot，换句话将，本地客户端在指定操作后不能得到服务器的即使反馈，这在FPS游戏中，会导致玩家的操作不够顺滑，因为Local Player操作后需要等待Server响应才能继续

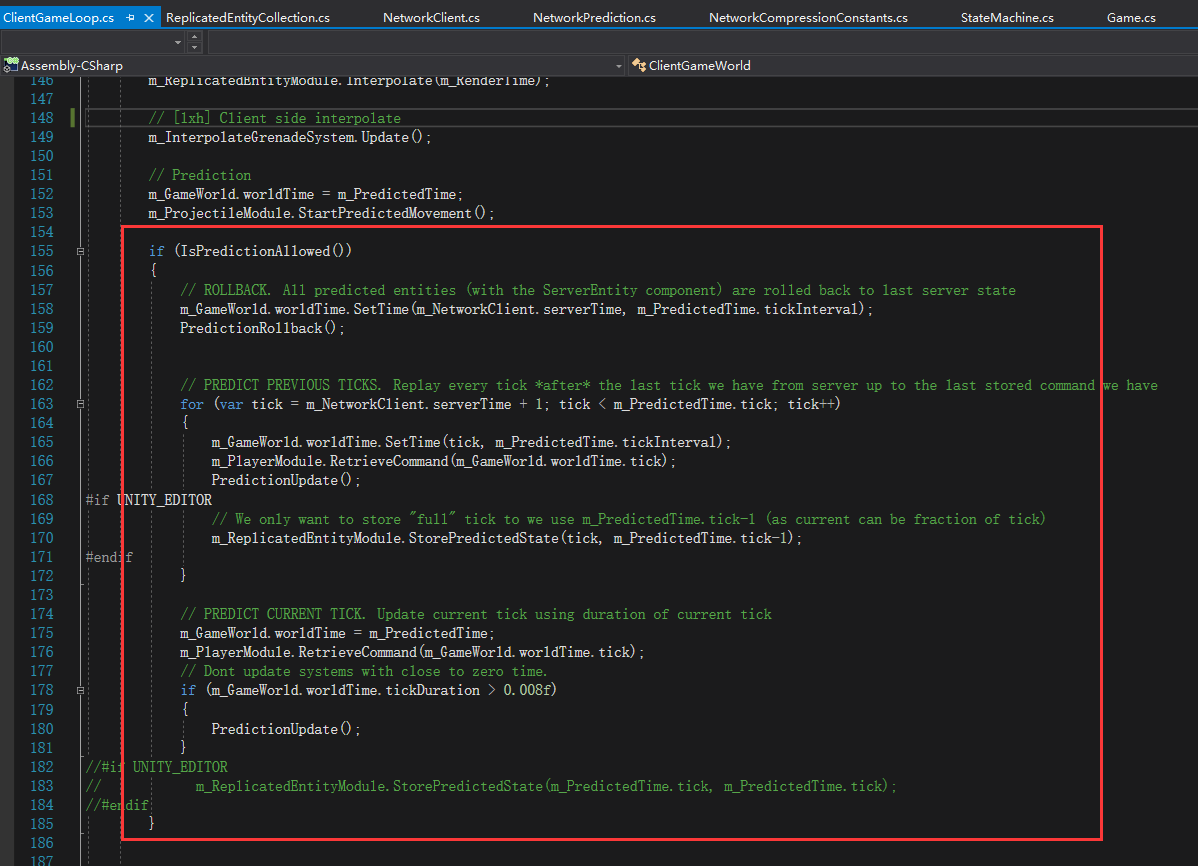
因此客户端对于Local Player采用’Client side prediction’





**Code Review**



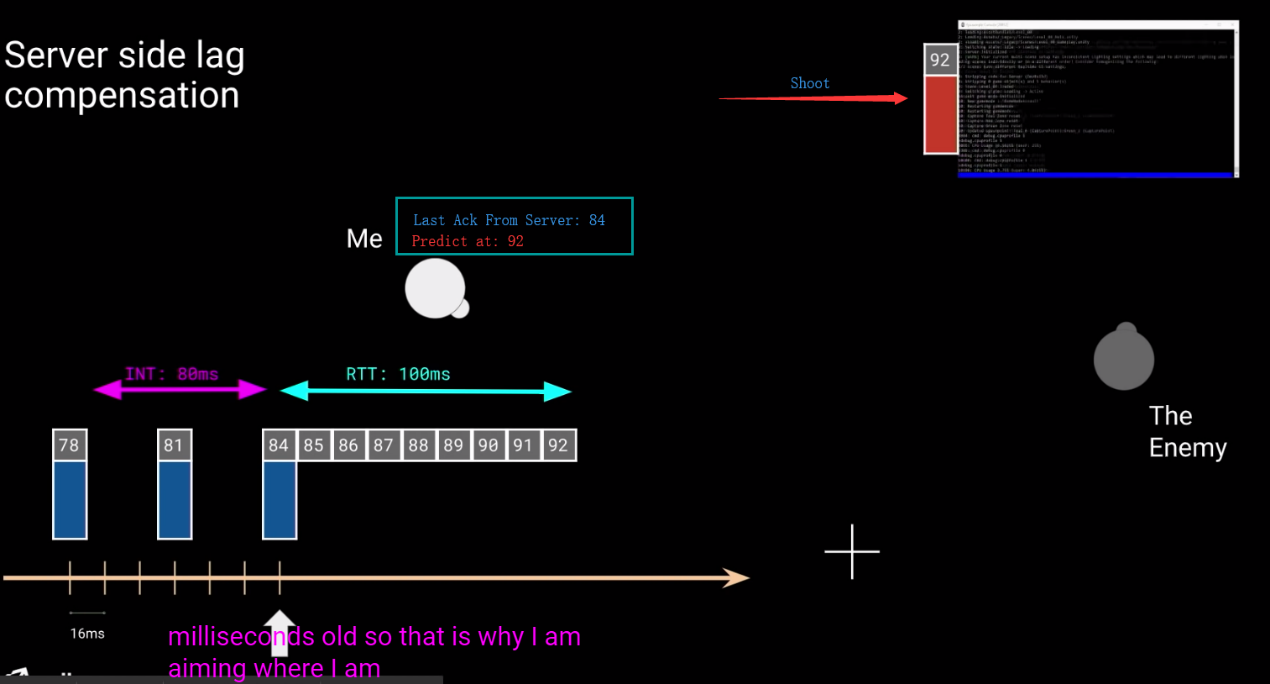


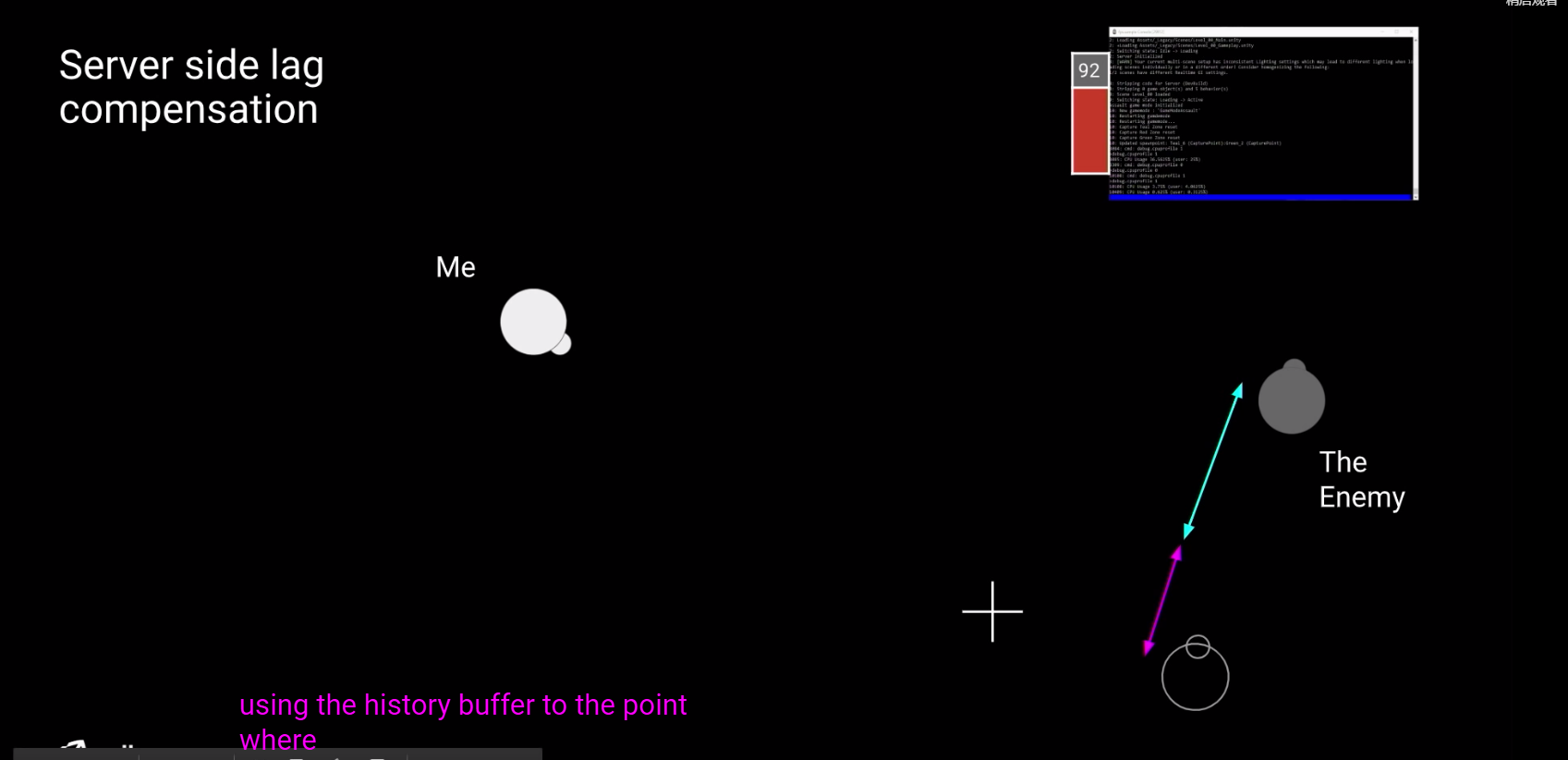
**6、Do I have to lead shots?** (\*)

Server side lag compensation

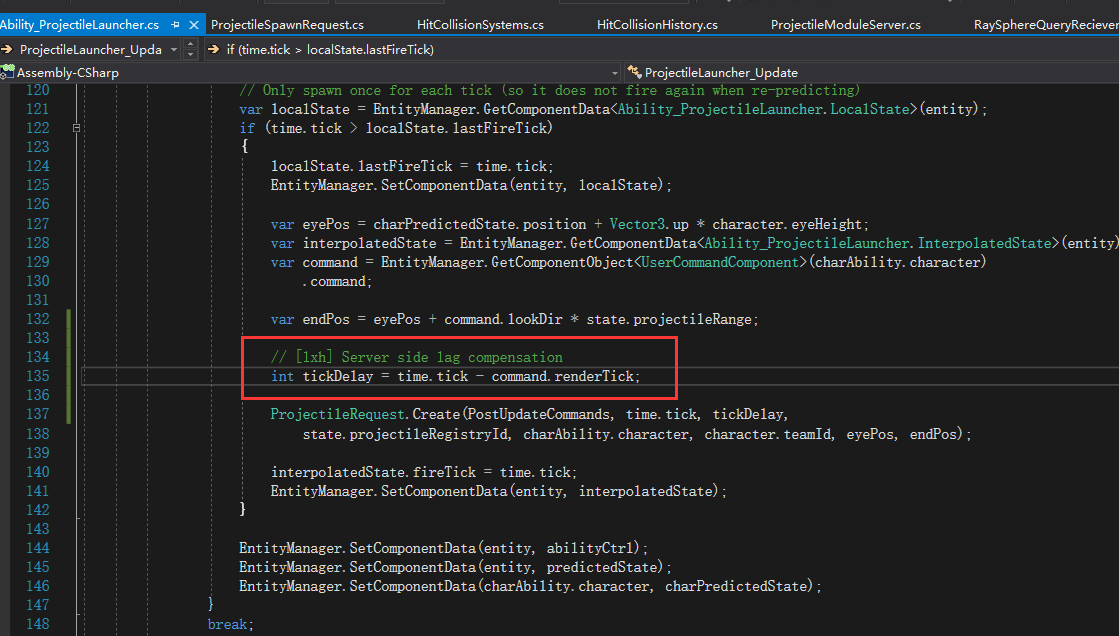
服务端保存最近128tick的Collision状态数据(HitCollisionHistory), 需要做命中检测时，服务端会将Collision状态(碰撞体的坐标和旋转状态)回退到射击客户端视角的Tick状态(RenderTick, UserCommand)

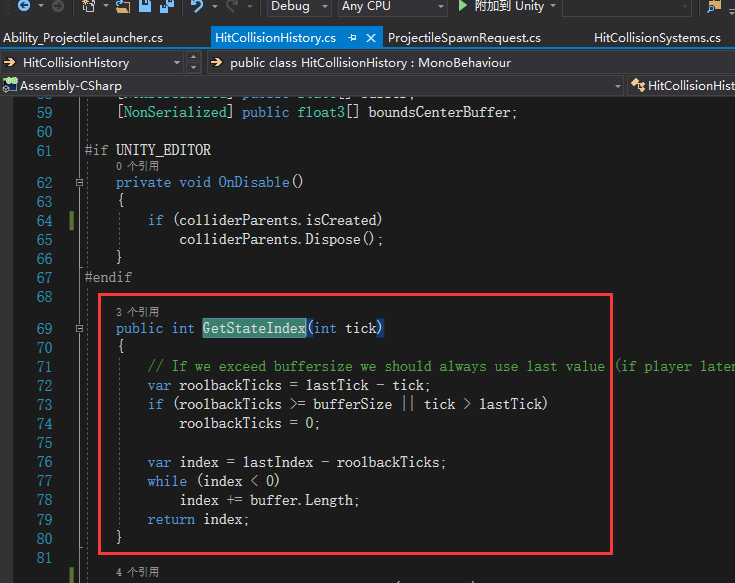
这样做一定的延时补偿，会大概率保证客户端的射击能准确命中

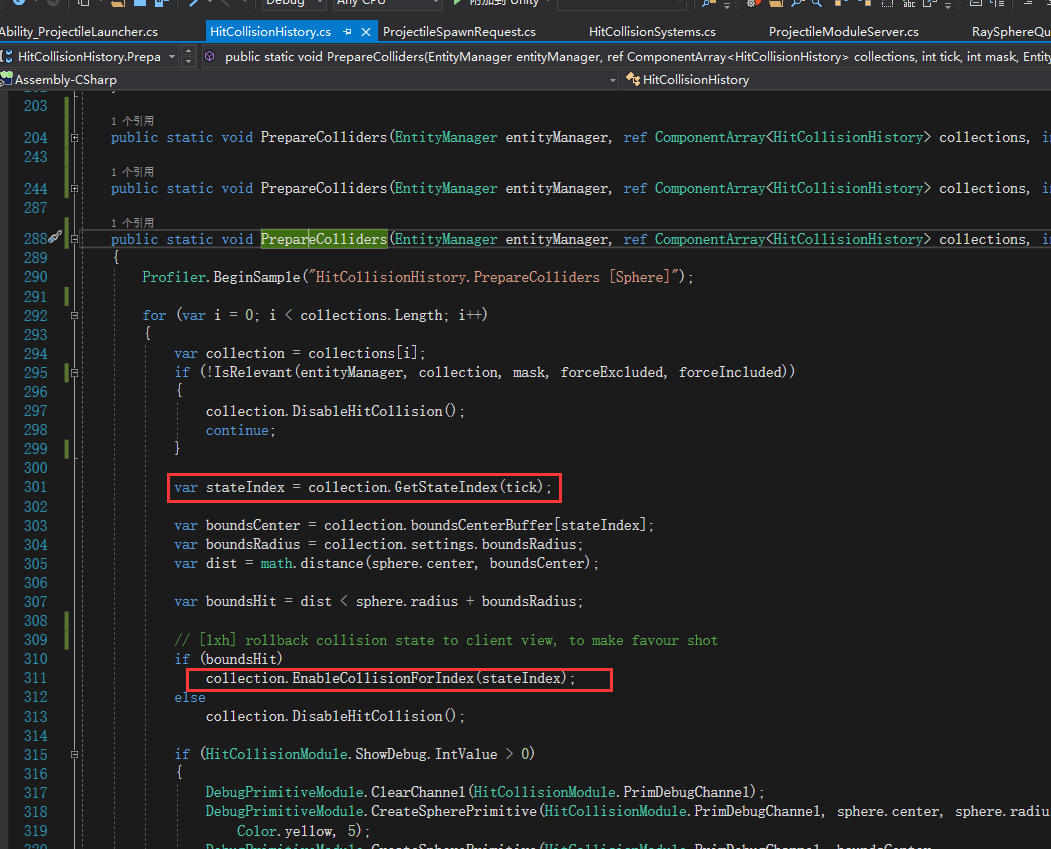




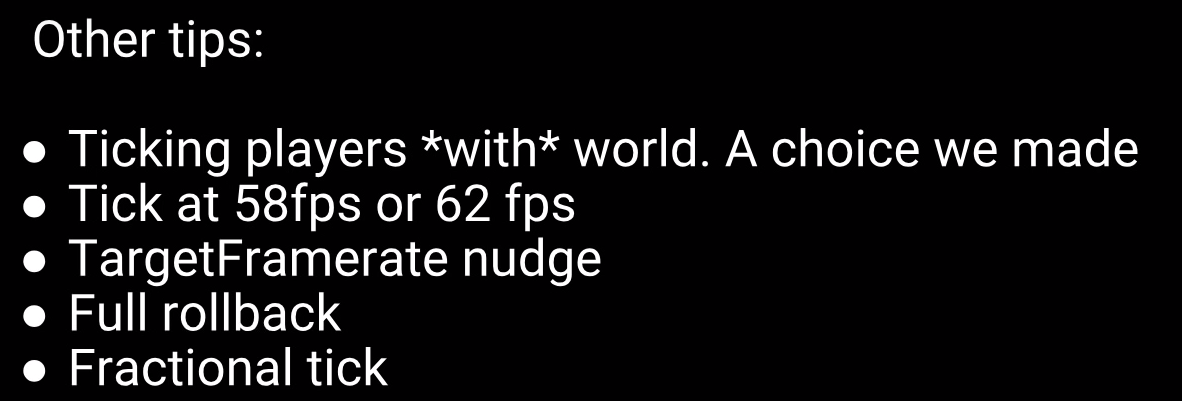
**Code View**







**7、Other tips**(\*)



<1> Ticking players with world

服务器缓存客户端的User Command Queue, 每次Simulate时直接Tick当前帧所有的User Command

<2> Tick at 58fps or 62

<3> TargetFramerate nudge

<4> Full rollback

<5> Fractional tick

Tip:(\*) means verified in code view; (\*) opposite