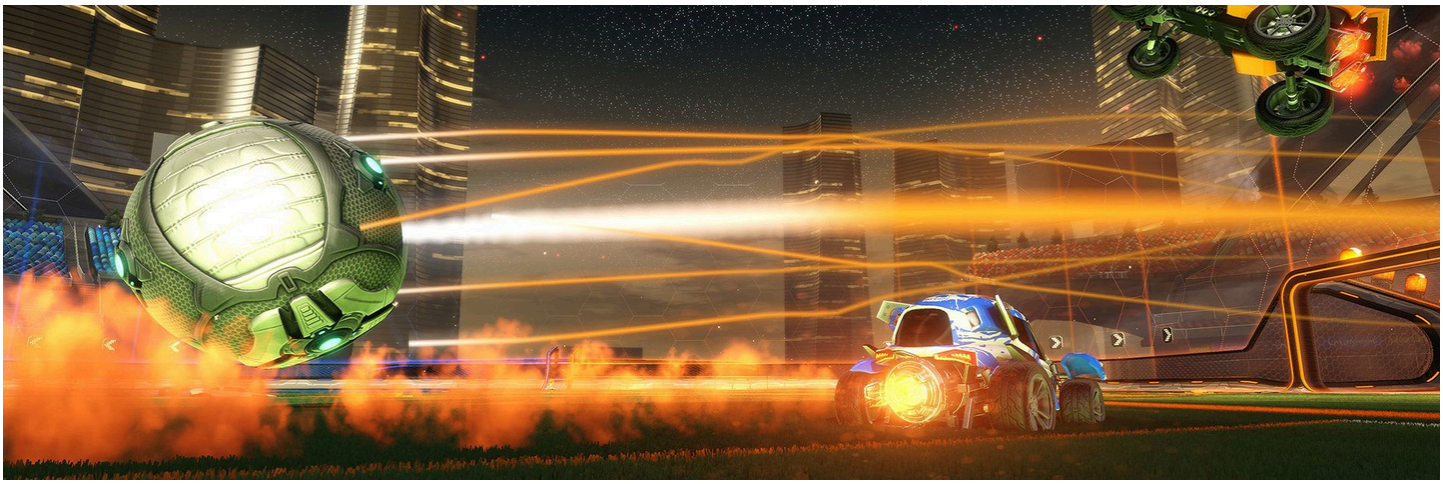


细谈网络同步在游戏历史中的发展变化（下）

网易游戏雷火事业群
已认证账号
已关注

编辑推荐

发布于 2020-12-15 14:46 · IP 属地浙江 ，编辑于 2023-05-25 15:02 · IP 属地浙江

终于完结啦！《细谈网络同步在游戏历史中的发展变化》系列文章已至尾篇。首篇细谈网络同步在游戏历史中的发展变化（上）我们讨论了网络同步的基本概念以及锁步同步（帧同步）的发展历史。上一篇文章细谈网络同步在游戏历史中的发展变化（中）我们分析了状态同步的发展历史以及相关优化手段，这篇主要是针对**物理同步**、**网络协议**以及**优化技术**三个方面做分析和总结。本文作者依旧是网易游戏雷火的游戏开发工程师@Jerish ，欢迎大家在评论区提问以及和我们互动！

> 前段时间，@韦易笑 老师阐述了有关“帧同步”一词在国内的发展历史也解释清了该名词乱用的原因。总的来说，国内“帧同步”与国外“Lockstep”的核心思想是一致的，为了方便描述，这篇文章会使用“帧同步”替代“LockStep”。

目录（下篇）：

五、物理|

1.概念与理解

所谓“物理同步”，字面上讲就是“带有物理状态对象的网络同步”，严格上来说它并不是一个标准的技术名词，而是大家约定俗成的一个概念。按照我的个人理解，可以进一步解释为**“在较为复杂的物理模拟环境或有物理引擎参与计算的游戏里，如何对持有物理状态信息的对象做网络同步”**。在英文中，我们可以使用Replicate physics-simulated objects 或者Networked physics来表示类似的概念。

不过，考虑到并不是所有物理现象都交给物理引擎处理，而且有物理引擎参与的网游也并不一定需要对同步做任何处理，所以我们常说的物理同步更多的是指“**在网络游戏中，如果玩家的位置或者与玩家交互对象的位置需要经过物理引擎的模拟处理来得到结果，那么其中涉及到网络同步技术就可以称为物理同步**”。（这里的物理模拟一般指整个对象完全交给物理引擎去计算碰撞、位置、约束等，很多情况下可以等价为对**Ragdoll**的模拟）

> 备注：物理一词涉及的范围非常广，在游戏里面应用的场景也很多，但是并不一定需要进行网络同步，比如简单的抛物线运动，射线检测，与玩法无关的场景破碎等。

早在上世纪70年代，就诞生了许多围绕物理特性产生玩法的游戏，不过由于当时计算机系统算力有限，涉及到的物理计算都非常简单（比如乒乓球游戏中小球的移动模拟[1]）。随着计算机性能的飞速提升，开发者们考虑将环境中的所有对象都交由统一的物理模块驱动，由此慢慢的催生出了通用的物理引擎[2]。很快的，各个游戏开发商逐渐将物理引擎集成进来，将更多更复杂的物理模拟过程应用到游戏中，制作出了诸如极品飞车、FIFA、NBA、愤怒的小鸟等围绕物理特性进行玩法设计的游戏。另一方面，随着计算机网络的发展，游戏中的网络同步技术愈加成熟，网络游戏的品质也不断向单机游戏靠拢，我们也得以将传统的单机游戏拓展成多人游戏。物理模拟作为提升游戏趣味性的一大技术也自然逐渐被纳入其中，物理同步变得重要起来。

2.物理同步面临的问题与解决方案

正如所前面解释的那样，物理同步并不是一种特殊的同步方式，而是在物理引擎和网络同步技术共同发展的条件下而诞生的一种综合行性解决方案，其核心手段还然是我们熟悉的帧同步或者状态同步。使用帧同步技术我们需要每帧把玩家的Input信息发送出去，然后让另一端的物理引擎根据输入去模拟结果。如果使用状态同步我们则需要本地模拟好数据并把物理位置、旋转等关键信息发送到其他的客户端，然后其他客户端可以根据情况决定是否再执行本地的物理模拟（如果是快照同步，由于拿到的就是最终的结果，那么就不需要本地再进行模拟了）。

这样看来，物理同步好像与常规的同步也没什么本质上的区别，那么为什么他却是一个难题呢？我认为原因有以下两点：

- 物理引擎的不确定性
- 在物理引

首先，我们谈谈**物理引擎的确定性问题**。很不幸，目前所有的物理引擎严格来说都不是确定性的，因为想保证不同平台、编译器、操作系统、编译版本的指令顺序以及浮点数精度完全一致几乎是不可能的。关于物理确定性的讨论有很多[3]，核心问题大致可以归类为以下几点：

1.编译器优化后的指令顺序

2.约束计算的顺序

3.不同版本、不同平台浮点数精度问题[4][5]

这里摘选一段PhysX物理引擎的描述[6]：

> The PhysX SDK can be described as offering limited determinism（注：提供了有限程度的确定性）. Results can vary between platforms due to differences in hardware maths precision and differences in how the compiler reoders instructions during optimization. This means that behavior can be different between different platforms, different compilers operating on the same platform or between optimized and unoptimized builds using the same compiler on the same platform（注：不同平台、编译器、优化版本都会影响确定性）. However, on a given platform, given the exact same sequence of events operating on the exact scene using a consistent time-stepping scheme, PhysX is expected to produce deterministic results. In order to achieve this determinism, the application must recreate the scene in the exact same order each time and insert the actors into a newly-created PxScene. There are several other factors that can affect determinism so if an inconsistent (e.g. variable) time-stepping scheme is used or if the application does not perform the same sequence of API calls on the same frames, the PhysX simulation can diverge.

如果游戏只是单个平台上发行，市面上常见的物理引擎（Havok，PhysX，Bullet）基本上都可以保证结果的一致性。因为我们可以通过使用同一个编译好的二进制文件、在完全相同的操作系统上运行来保证指令顺序并解决浮点数精度问题，同时打开引擎的确定性开关来保证约束的计算顺序（不过会影响性能），这也是很多测试者在使用Unity等商业引擎时发现物理同步可以完美进行的原因。当然，这并不是说我们就完全放弃了跨平台确定性的目标，比如Unity新推出的DOTS架构[7][8]正在尝试解决这个问题（虽然注释里面仍然鲜明的写着“Reserved for future”）。

考虑到物理引擎的确定性问题，我们可以得出一个初步的结论——完全使用帧同步做物理同步是不合适的（或者说做跨平台游戏是行不通的）。而对于状态同步，我们可以定时地去纠正位置信息来避免误差被放大。如果一定要使用帧同步去做跨平台同步，那么只能选择放弃物理引擎自己模拟或者用定点数来改造物理引擎，这可能是得不偿失的。

下面不妨先排除掉一致性的问题，来看看如何实现所谓的“物理同步”。实际上，无论是优化手段还是实现方式与前两篇提到的方案是几乎一致的，帧同步、快照同步、状态同步都可以采用，增量压缩、Inputbuffer等优化手段也一样可以用于物理同步的开发中。Network Next的创始人Glenn Fiedler在2014年撰写了一系列的物理同步相关的文章[9]，使用一个同步的Demo非常详细地阐述了同步技术是如何应用以及优化的，涉及到的技术点大致如下，涵盖了网络同步的大部分的知识细节：

- 如何确保物理引擎的确定性
- 如何实现物理帧同步
- Inputbuffer如何改善帧同步
- 为什么用UDP替代TCP
- 如何实现快照同步
- 怎样用插值解决网络抖动
- 如何通过快照压缩减少网络流量
- 如何实现增量压缩
- 如何实现状态同步

另外，在2018年的GDC上，Glenn也对物理同步进行一次演讲分享[10]，具体的细节建议大家移步到Glenn Fiedler的网站以及GitHub[11]去看。

接下来，我们再来谈谈第二个难点，即**网络同步的误差是如何被物理模拟迅速放大的（尤其在多人交互的游戏中）**。我们在前面的章节里也谈过，为了保证本地客户端的快速响应，通常会采取预测回滚的机制（Client prediction，即本地客户端立刻相应玩家操作，服务器后续校验决定是否合法）。这样我们就牺牲了事件顺序的严格一致来换取主控端玩家及时响应的体验，在一般角色的非物理移动同步时，预测以及回滚都是相对容易的，延迟比较小的情况位置的误差也可以几乎忽略。然而在物理模拟参与的时候，情况就会变得复杂起来。

> 主控（Autonomous/Master）即当前角色是由本地玩家控制的，模拟（Simulate/Replica）即当前角色是由其他玩家控制的

假如在一个游戏中（带有预测，也就是你本地的对象一定快于远端）你和其他玩家分别控制一个物理模拟的小车朝向对方冲去，他们相互之间可能发生碰撞而彼此影响运动状态，就会面临下面的问题。

1.由于网络同步的误差无法避免，那么你客户端上的发生碰撞的位置一定与其他客户端的不同。

2.其次，对于本地上的其他模拟小车，要考虑是否在碰撞时完全开启物理模拟（Ragdoll）。如果不开启物理，那么模拟小车就会完全按照其主控端同步的位置进行移动，即使已经在本地发生了碰撞他可能还是会向前移动。如果开启碰撞，两个客户端的发生碰撞的位置会完全不同。无论是哪种情况，网络同步的误差都会在物理引擎的“加持”下迅速被放大进而导致两端的结果相差甚远。

其实对于一般角色的非物理移动同步，二者只要相撞就会迅速停止移动，即使发生穿透只要做简单的位置“回滚”即可。然而在物理模拟参与的时候，直接作位置回滚的效果会显得非常突兀并出现很强的拉扯感，因为我们几乎没办法在本地准确的预测一个对象的物理模拟路径。如果你仔细阅读了前面Glenn Fiedler的文章（或者上面总结的技术点），你会发现里面并没有提到常见的预测回滚技术，因为他只有一个主控端和一个用于观察结果的模拟端，并不需要回滚。

在2017年的GDC上，来自育碧的技术负责人Matt Delbosc就《看门狗2》中的载具同步进行了演讲[12]，详细的分析了多个主控端控制不同对象发生碰撞时应该如何处理。

《看门狗2》的网络模型是基于状态同步的P2P，主控角色预测先行而模拟对象会根据快照（snapshot，即模拟对象在其主控端的真实位置）使用Projective Velocity Blengding做内插值，他们在制作时也面临和上面描述一样的问题。假如两个客户端各控制一个小车撞向对方，由于延迟问题，敌人在本地的位置一定是落后其主控端的。那么就可能发生你开车去撞他时，你本地撞到了他的车尾，而他的客户端什么都没有发生。

所以，首先要做的就是尽量减少不同客户端由于延迟造成的位置偏差，Matt Delbosc引入了一个TimeOffset的概念，根据当前时间与TimeOffset的差值来决定对模拟对象做内插值还是外插值，有了合适的外插值后本地的模拟对象就可以做到尽量靠近敌方的真实位置。

而关于碰撞后位置的误差问题，他们采用了Physics Simulation Blending技术，即发生碰撞前开启模拟对象的RigidBody并设置位置权重为1（快照位置的权重为0），然后在碰撞发生后的一小段时间内，不断减小物理模拟的权重增大快照位置的权重使模拟对象的运动状态逐渐趋于与其主控端，最终消除不一致性，腾讯的吃鸡手游就采用了相似的解决方案[13]。

不过实际上， Matt团队遇到的问题远不止这些，还有诸如如何用插值解决旋转抖动问题，人物与载具相撞时不同步怎么办等等，知乎上有一篇译文可以参考[14]。

> 可能有些朋友会问，如果我不使用预测回滚技术是不是就没有这个问题呢？答案依然是否定的，假如你在运行一个车辆的中间突然变向，而这个操作被丢包或延迟，只要服务器不暂停整个游戏来等待你的消息，那么你本地的结果依然与其他客户端不同进而产生误差。也就是说除非你使用最最原始的“完全帧同步”（即客户端每次行动都要等到其他客户端的消息全部就绪才行），否则由于网络同步的延迟无法避免，误差也必定会被物理模拟所放大

同样在2017年，另一款风靡全球的竞技游戏——《火箭联盟》悄然上线，可谓是将物理玩法发挥到了极致。次年，《火箭联盟》的开发者Jared Cone也来到了GDC，分享了他们团队是如何解决物理同步问题的[14]。

《火箭联盟》的核心玩法是“用车踢球”，每个玩家控制一个汽车，通过撞击足球来将其“踢”进敌方的球门。由于是多人竞技游戏，所以一定要有一个权威服务器来避免作弊，最终的结果必须由服务器来决定。相比于《看门狗》，他们遇到的情况明显更复杂，除了不同玩家控制不同的小车，还有一个完全由服务器操控的小球。按照常规的同步方式，本地的主控玩家预测先行，其他角色的数据由服务器同步下发做插值模拟。但是在这样一个延迟敏感且带有物理模拟的竞技游戏中，玩家的Input信息的丢失、本地对象与服务器的位置不统一都会频繁的带来表现不一致的问题，而且FPS中常见的延迟补偿策略并不适合当前的游戏类型（简单来说就是延迟大的玩家会影响其他玩家的体验，具体原因我们在上一篇延迟补偿的章节也有讨论）。

为了解决这些问题，Jared Cone团队采用了“InputBuffer”以及“客户端全预测”两个核心方案。InputBuffer，即服务器缓存客户端的Input信息，然后定时的去buffer里面获取（buffer大小可以动态调整），这样可以减少网络延迟和抖动带来的卡顿问题。

客户端全预测，即客户端上所有可能产生移动的对象（不仅仅是主控对象）全部会在本地预测先行，这样本地在预测成功时所有对象的位置都是准确的，客户端与服务器的表现也会高度一致，当然预测失败的时候自然会也要处理位置回滚。

仔细分析这两款游戏，你会发现他们采用都是“状态同步+插值+预测回滚”的基本框架，这也是目前业内上比较合适的物理同步方案。

除了同步问题，物理引擎本身对系统资源（CPU/GPU）的消耗也很大。比如在UE4引擎里面，玩家每一帧的移动都会触发物理引擎的射线检测来判断位置是否合法，一旦场景内的角色数量增多，物理引擎的计算量也会随之增大，进而改变Tick的步长，帧率降低。而帧率降低除了导致卡顿问题外，还会进一步影响到物理模拟，造成更严重的结果不一致、模型穿透等问题，所以我们需要尽量减少不必要的物理模拟并适当简化我们的计算模型。

六、TCP VS UDP

网络同步本质是数据的传输，当逻辑层面优化已经不能满足游戏的即时性要求时，我们就不得不考虑更深一层协议上的优化，而这件事开发者们从上世纪90年代就开始尝试了。

按照OSI模型（**O**pen **S**ystem **I**nterconnection Model），我们可以将计算机网络分为七层。一般来说，我们在软件层面（游戏开发）最多能干涉的到协议就是传输层协议了，即选择TCP还是UDP。网上关于TCP和UDP的文章与讨论有很多[15]，这里会再帮大家梳理一下。

TCP（**T**ransmission **C**ontrol **P**rotocol），即传输控制协议，是一种面向连接的、可靠的、基于字节流的传输层通信协议[16]。该协议早在1974年就被提出并被写进**RFC**（**R**equest for **C**omments）中，自发布几十年来一直被不断优化和调整，如今已经是一个包含“可靠传输”，“拥塞控制”等多个功能的协议了（RFC 2581中增加）。

在21世纪早期，我们因特网上的数据包有大约95%的包都使用了TCP协议，包括HTTP/HTTPS，SMTP/POP3/IMAP、FTP等。当然，也包括一大部分网络游戏。我们如此偏爱于TCP就是因为他从协议层面上提供了许多非常重要的特性，如数据的可靠性、拥塞控制、流量控制等。这些可以让软件应用的开发者们无需担心数据丢失、重传等细节问题。

然而在游戏开发中，这些特性却可能是网络同步中的负担。在FPS游戏中，玩家每帧都在移动，我们期望这些数据在几毫秒内就能送达，否则就会对玩家产生干扰、影响游戏体验。因此对于FPS、RTS这种要求及时响应的游戏，TCP协议那些复杂的机制看起来确实有点华而不实。

考虑到TCP协议非常复杂，这里只从几个关键的点来谈谈他的问题[17]。

1.在TCP中，数据是通过字节流的方式发送的，但由于建立在IP协议上必须将字节流拆分成不同的包，默认情况下协议会将你的数据包缓冲，到达一定值才会发送。这样可能会出现在游戏某个阶段你最后几个包明明已经执行了发送逻辑，但是由于缓冲机制被限制而无法到达。不过好在我们可以通过TCP_NODELAY来设置TCP立即刷新写入它的所有数据。

2.其次，TCP的可靠数据传输响应并不及时。一旦数据包发生丢失或乱序，那么接收方就会一直等待这个数据的到来，其他新收到的数据只会被缓存在协议层，你在应用层根本获取不到任何数据也无法做任何处理。这个时候你可能要等超时重传机制响应后才能拿到重发的数据包，这时候可能已经过了几十毫秒。即使TCP拥有快速重传机制，仍然达不到理想的延迟效果。

3.拥塞控制和流量控制不可控。TCP在网络环境比较差的条件下，会通过调整拥塞控制窗口大小来减少包的发送来降低吞吐量，这对于延迟敏感的游戏完全是无法接受的。同样，我们在应用层上面也无能为力。

4.其他的还有一些的小问题，比如每个TCP的报头都需要包含序列号、确认号、窗口等数据结构，无形中增加了流量大小；TCP需要在端系统中维护连接状态，包括接收与发送缓存、拥塞控制参数等，在处理大量连接的消息时也更为繁琐和耗时。

那么这些问题能解决么？也许能，但是从协议层面我们无能为力，因为TCP协议设计之初就不是为了及时响应，而另一个运输层协议UDP看起来比较符合我们的理念。

UDP(**U**ser **D**atagram **P**rotocol)，即用户数据包协议，是一个简单的面向数据报通信的协议[18]。该协议由David P. Reed在1980年设计并写入RFC 768中。顾名思义，UDP设计之初就是为了让用户可以自由的定义和传输数据，不需要建立链接、没有流量控制也没有拥塞控制，但是会尽可能快的将数据传输到目的IP和端口。

在上世纪90年代，Quake等游戏就开始使用UDP协议取代TCP进行数据同步，结果也很理想。除了游戏外，其他诸如视频、语音通信等领域也在广泛使用UDP，开发者们开始基于UDP创建自定义的Reliable UDP通信框架（QUIC、WbRTC、KCP、UDT等[19]），一些游戏引擎（如UE4）也将RUDP集成进来。随着网络带宽的提高，使用UDP代替TCP目测是一个趋势（参考Http3[20]）。

虽然UDP很自由，但是需要开发者们自己写代码完善他。我们需要自己去写服务器客户端建立链接的流程，我们需要手动将数据分包，我们还需要自己实现应用层面的可靠数据传输机制。另外，UDP还有一个传输上的小劣势——当路由器上的队列已满时，路由器可以根据优先级决策在丢弃TCP数据包前先丢失UDP，因为他知道TCP数据丢失后仍然会进行重传。

总的来说，对于那些对延迟很敏感的游戏，UDP的传输模式更加适合而且弹性很大，同时他也可以胜任那些同步频率比较低的游戏，但是UDP的开发难度比较高，如果是自己从零开发确实有相当多的细节需要考虑，所以建议大家在已有的RUDP框架上进行优化。

七、常见同步优化技术

梳理完同步的发展历史，我们最后再来总结一下常见的网络同步优化技术。首先，提出一个问题，网络同步优化到底是在优化什么？

在单机游戏中，从我们按下按键到画面的响应中间经历了输入采样延迟、渲染流水线、刷新延迟、显示延迟等。而在一个网络游戏中，从我们按下按键到另一个机器收到指令，则会经历一个极为耗时的网络延迟（相比之下，单机的延迟可以忽略不计）。网络延迟其实也包括处理延迟、传输延迟（主要延迟）、排队延迟以及传播延迟，一般我们会将这些延迟统称为网络延迟，我们优化的目的就是想尽各种办法**降低或是抵消掉这个延迟**。

数据从客户端传输到服务器的一个来回称为一个RTT。在CS架构下，其实每个客户端的行为一直是领先于服务器1/2个RTT的，数据从客户端发送到服务器有一个1/2的RTT延迟，服务器处理后通知客户端又有一个1/2的RTT延迟。P2P架构下，由于没有权威服务器，我们可以省去1/2的 RTT延迟，但是在目前的网络游戏中，为了对抗作弊行为以及容纳更多的玩家，我们不得不采用CS架构。

由于在网络游戏中，延迟是不可避免的，所以我们的优化手段就是如何减小这个延迟以及如何让玩家感受不到延迟。下面我会从**表现优化、延迟对抗、丢包对抗、带宽优化以及帧率优化**这几个方面来做一下总结，

1.表现优化（弱化玩家对延迟的感受）：

a.插值优化

通过内插值解决客户端信息离散更新的突变问题，通过外插值解决网络延迟过大收不到数据而卡顿的问题，两种方案并不冲突，可以同时采用。在具体应用时，我们可以使逻辑帧与渲染帧分离，这样在客户端没有收到数据的时候还可以继续更新渲染，也可以在二者不分离的情况只对渲染的模型进行插值，客户端收到权威数据后再进行整个对象的移动。

b.客户端预先执行+回滚

预测的目的是让玩家能在本地操作后立刻收到反馈，提升游戏体验，回滚是为了保证服务器的权威性。客户端预测包括位置预测以及行为预测两种，位置预测需要高频率的执行，因为移动在每帧都可能发生，而其他行为预测则相对低频一些，包括开枪、扔手雷、释放技能等。另外，对于延迟不太敏感的游戏（比如MMO），可以放宽校验条件（超过一定误差再纠正），这样即使降低服务器帧率客户端也不会有什么感觉。

2.延迟对抗（弱化玩家对延迟的感受）：

a.延迟补偿

服务器记录一段时间内所有玩家的位置历史，在发生伤害计算时根据延迟对所有玩家角色进行位置的回滚与处理，可以尽量还原当时的场景。

b.命令缓冲区

把远端的数据缓存在一个buffer里面，然后按照固定频率从buffer里面取，可以解决客户端卡顿以及网络抖动问题。不过缓冲区与延迟是有冲突的，缓冲区越大，证明我们缓存的远端数据就越多，延迟就越大。

c.从具体实现的技巧上对抗延迟

操作加一个前幺时间，客户端释放技能等行为前有一个时间来抵消掉RTT的延迟。如无敌状态做一个过度动画，客户端播放动画后进入无敌，但是服务器可以在收到指令后直接进入无敌状态从而抵消延迟。在游戏Halo中，有很多类似的例子。比如在客户端玩家扔手雷的时候，我们可以在本地立刻播放扔手雷的动画并发送请求到服务器，然后服务器收到后不需要播放动画立刻生成手雷并同步，这样客户端真正扔出手雷的表现就是0延迟的。

3.丢包对抗（弱化玩家对延迟的感受）：

a.使用TCP而不是UDP

由于TCP不会丢包，对于延迟不敏感的游戏还是优先采取TCP来对抗丢包

b.冗余UDP数据包

一次性发送多个帧的数据来对抗丢包。UDP同步数据时经常容易丢包，我们虽然可以使用上层实现UDP的可靠性，但是像帧同步这种同步数据量比较小的游戏可以采用冗余UDP的方案，即后续的UDP包会冗余一定量的前面已发送的UDP包，这样即使丢失了部分包我们也能保证拿到完整的远端数据。

4.带宽优化（减小延迟）：

带宽优化的目的是减小客户端以及服务器的同步压力，避免大量数据同时传输造成处理不过来，排队甚至是丢失。带宽优化是非常灵活且多变的，我们需要根据游戏的玩法来调整我们的优化行为。

a.同步对象裁剪

核心目的是根据相关性剔除那些不需要同步的对象（这里都是指在同一个服务器内），比如一个玩家距离我很远，我们的行为彼此不会影响，所以就不需要互相同步对方的数据。裁剪方式有非常多，常见的SOI（Spheres of Influence），静态区域（把场景划分为N个小区域，不在一个区域不同步），视锥裁剪（更多用于渲染），八叉树裁剪等。相关性还可能会涉及到声音等其他因素，需要根据自己项目来决定

这里着重提一点AOI（Area Of Interest [21][22]），即根据玩家的位置维护一个动态的视野列表，视野外的对象会被完全忽略（能大幅的减少同步对象的遍历与比较）。其基本思想也是判断相关性，实现方式有很多，其中基于格子的空间划分算法是网络游戏中常见的实现方案。在虚幻引擎中，大世界同步框架ReplicationGraph[23]的核心思想也是如此。不过要注意的是，对于MMO这种可能有大量角色同时进行连续移动的游戏，视野列表频繁的增删查操作也可能对服务器造成一定的压力。

b.分区，分房间

对于大型MMO来说，这是常见的手段，将不同的玩家分散到不同的场景内（不同的服务器），这样减小服务器处理数据的压力，减小延迟。对于大世界游戏而言，不同服务器可能接管同一个地图不同区域的服务，其中的跨服数据同步比较复杂。

c.数据压缩与裁剪

坐标与旋转是我们常见的同步内容，但是很多数据其实是不需要同步的。比如对于大部分3D游戏角色的Pitch以及Roll是不会改变的，我们只要同步Yaw值即可。对于非第一人称游戏，我们可以接着把四个字节float类型的Yaw压缩到两个字节的uint16里面，玩家根本不会有什么体验上的差异。类似的方法可以应用到各种同步数据里面。

此外，在状态同步里面，我们可以采用增量发送来减少数据量，即第一次发送完整的数据信息后只发送哪些发生过变化的数据，这可以大大减少网络同步的流量。

d.减少遍历以及更细力度的优化

在Halo以及虚幻引擎里面都会对同步对象做优先级划分，发送频率调整等。在状态同步中，我们还需要合适的手段来快速定位发生变化的数据，如属性置脏、利用发射减少非同步属性的遍历等。进一步的，我们还可以根据客户端的类型以及信息作出更细致的同步信息过滤以及设置优先级，比如对同步属性进行优先级划分等（目前还没有见到过粒度如此细致的，但理论上是可行的）。

5.帧率优化（减小延迟）：

帧率优化是一个重要且复杂的难题，涉及到方方面面的技术细节，这里主要针对网络同步相关内容做一些分析。相比单机游戏，网游需要同时考虑客户端与服务器的帧率，这并不是单纯地提升帧率的问题，如何优化与平衡是一个很微妙的过程。

a.提升帧率

这个不用多说，帧率低就意味着卡顿，玩家的体验就会很差。不同游戏的性能瓶颈都可能不一样，包括内存问题（GC、频繁的申请与释放）、IO（资源加载、频繁的读写文件，网络包发送频率过大，数据库读取频繁）、逻辑问题（大量的遍历循环，无意义的Tick，频繁的创建删除对象，过多的加锁，高频率的Log）、AI（寻路耗时[24]）、物理问题（复杂模拟，碰撞次数过多）、语言特性（脚本语言比较费时）等，客户端相比服务器还有各种复杂的渲染问题（Drawcall太多，半透明，动态阴影等）。这些问题需要长期的测试与调试，每个问题涉及到的具体细节可能都有所不同，需要对

b.保持帧率稳定与匹配

假如你的客户端与服务器帧率已经优化到极致，你也不能任其自由变化。首先，要尽量保持服务器的帧率稳定（减少甚至是消除玩家比赛时的所有潜在的卡顿问题），考虑一款对延迟比较敏感的射击游戏，如果你的客户端在开枪时遇到了服务器卡顿，那么就可能造成校验失败，导致客户端看到的行为与服务器行为不一致。其次，还要保持客户端与服务器的帧率匹配。对于延迟不敏感的游戏，考虑到玩家的体验以及服务器的压力，客户端的帧率可以高于服务器多倍，但是这个比例是需要通过实际的测试来调整。而对于延迟敏感的游戏，我们一般需要尽量让服务器的帧率接近客户端，这样服务器才能更及时的相应，减少延迟带来的误差。此外，我们也不能让客户端的帧率无限提高，对于某些同步算法，客户端与服务器过高的帧率差异可能造成不断的拉回卡顿。所以，很多游戏会采取锁帧的方式来保证游戏的稳定性。

c.计算压力分担

对于MMO这种服务器压力比较大的游戏，我们通常会考虑把一部分计算资源转交给客户端去计算（甚至是计算后再返还给服务器），比如物理运算、自动寻路、AI逻辑计算等。其实将这种方式使用到极致的例子就是帧同步，服务器只做一些简单的校验即可。

总的来说，网络同步优化是一个长期的不断试错的过程，我们需要合理的利用计算机资源，把最重要的资源用在最重要的功能上面，减少重复的计算与流程，并需要配合一些经验和技巧来规避那些不好解决的问题。

八、总结

我们从最开始的网络游戏架构谈起，按照时间线梳理了近几十年“帧同步”与“状态同步”的发展历程，并讲述了各种同步技术以及优化方案。虽然网络同步是游戏中的技术，但其本质还是计算机数据的同步。无论是Lockstep还是TimeWarp，最初都是用于计算机系统通信的技术，只不过应用场景从一台机器的内部通信转变为多台机器的通信，从传统的应用转移到网络游戏上面。

游戏的类型会影响到网络同步的解决方案，也会影响到项目的整体架构，所以我们在制作一款网络游戏前要事先做好需求分析并确定网络同步方案。同时也要意识到，网络同步延迟是不可消除的，除了算法层面的优化外还可以从实现技巧上来规避一些难题。

到此，历时半年多的网络同步系列终于迎来完结。不过网络技术还在进步，历史也还在前行，让我们一同关注同步技术的发展和变化，期待未来的游戏世界。

[1] WIKI, "Pong", WIKI, 2020.Available:en.wikipedia.org/wiki/P... [Accessed:2020-12-12]

[2] Tony Wang, "游戏物理模拟简史", 知乎, 2020.Available:zhuanlan.zhihu.com/p/10... [Accessed:2020-12-12]

[3] Theraot, "How can I perform a deterministic physics simulation?",Gamedev Stackexchange, 2019.gamedev.stackexchange.com... [Accessed:2020-12-12]

[4] Yossi Kreinin, "Consistency: how to defeat the purpose of IEEE floating point", Personal Blog , 2008. Available:yosefk.com/blog/consist... [Accessed:2020-12-12]

[5] Glenn Fiedler, "Floating Point Determinism", Personal Blog , 2010. Available:gafferongames.com/post/... [Accessed:2020-12-12]

[6] NVIDIA, "NVIDIA PhysX SDK 3.4.0 Documentation Determinism", NVIDIA , 2020. Available:docs.nvidia.com/gamewor... [Accessed:2020-12-12]

[7] MelvMay, "How much deterministic is Physics from Unity3d In 2019?",Unity Forum,2020.Available:

forum.unity.com/threads... [Accessed:2020-12-12]

[8] Unity, "Burst User Guide",Unity Manual, 2020.Available:docs.unity3d.com/Packag... [Accessed:2020-12-12]

[9] Glenn Fiedler, "Introduction to Networked Physics", Personal Blog， 2014. Available: gafferongames.com/post/...[Accessed:2020-12-12]

[10] Glenn Fiedler， "Physics for Game Programmers : Networking for Physics Programmers"， 2018.Available: gdcvault.com/play/10221...[Accessed:2020-12-12]

[11] Glenn Fiedler， "UnityDemo: Networked Physics in Virtual Reality: Networking a stack of cubes with Unity and PhysX" , 2018. Available: github.com/fbsamples/oc... [Accessed:2020-12-12]

[12] Matt Delbosc, "Replicating Chaos Vehicle Replication in Watch Dogs 2", GDC, 2017. Available: bilibili.com/video/BV1K... [Accessed:2020-12-12]

[13] Ned,"手游中载具物理同步的实现方案", 腾讯游戏学院, 2018. Available: gameinstitute.qq.com/kn... [Accessed:2020-12-12]

[14] Funny David, "看门狗2的载具同步(翻译)", 知乎, 2019. Available: zhuanlan.zhihu.com/p/95... [Accessed:2020-12-12]

[15] Jared Cone, "It IS Rocket Science! The Physics of 'Rocket League' Detailed", GDC, 2018. Available: bilibili.com/video/av44... [Accessed:2020-12-12]

[16] Glenn Fiedler, UDP vs. TCP gafferongames.com/post/... [Accessed:2020-12-12]

[17] WIKI, "Transmission Control Protocol", WIKI, 2020. Available: en.wikipedia.org/wiki/T... [Accessed:2020-12-12]

[18] Draveness, "为什么 TCP 协议有性能问题", Personal Blog, 2020. Available: draveness.me/whys-the-d... [Accessed:2020-12-12]

[19] WIKI, "User Datagram Protocol", WIKI, 2020. Available: en.wikipedia.org/wiki/U... [Accessed:2020-12-12]

[20] 小玩童， "Reliable UDP一览：那些能替代TCP的RUDP方案", Personal Blog, 2020. Available: juejin.cn/post/68449040...[Accessed:2020-12-12]

[21] WIKI, "HTTP/3", WIKI, 2020. Available: en.wikipedia.org/wiki/H... [Accessed:2020-12-12]

[22] 哈库纳, "聊一聊游戏服务器架构设计－聊天功能的那些事" ,Personal Blog,2016.Available: my.oschina.net/ta8210/b...[Accessed:2020-12-12]

[23] 云风, “AOI服务的设计与实现”,Personal Blog,2012. Available: blog.codingnow.com/2012... [Accessed:2020-12-12]

[24] Jerry, zhuanlan.zl

[25] 王杰, "揭秘重度MMORPG手游后台性能优化方案",知乎,2018. Available: zhuanlan.zhihu.com/p/49...[Accessed:2020-12-12]

发布于 2020-12-15 14:46 · IP 属地浙江 ， 编辑于 2023-05-25 15:02 · IP 属地浙江

游戏开发 网络游戏 游戏

理性发言，友善互动

22 条评论 默认 最新

meta
有点浮于表面了，udp的应用场景需要深入的讨论，而且使用udp需要稍微考虑一下流控问题。
2021-01-18 回复 4

meta
有干货的也就是这一篇，但是你们调用gdc实际上并不全，优秀的关于同步的文章很多。
2021-01-18 回复 3

根号三
给逆水寒换个策划吧！
2021-01-08 回复 4

小钱
完全看不懂说啥，细节一点都没有透露，纯属炫技
2021-06-28 回复 3

千丘禾
细节都没透露，炫了个啥技啊😳
2023-11-13 回复 喜欢

仰泳的鱼
给同事点赞😘😘😘
2020-12-28 回复 1

逸风
看到雷火这两个字就觉得恶心
2020-12-19 回复 1

无冬
没人逼你看
2021-01-04 回复 3

埃玛
到了PUBG，apex，ow和永劫无间的时代回合频率越来越快，需要处理的逻辑越来越复杂，现有的同步技术是不是不够用了
2023-02-09 回复 喜欢

飞的飞
浮点数改成定点数会有哪些苦难呢，我记得腾讯有本书里面讲过这个技术
2021-11-02 回复 喜欢

丐刘
对于帧同步不适合做物理同步的原因，是指不同平台的因素会导致不一致吗
2021-10-21 回复 喜欢

当当当

想问问荒野行动中的物理同步是怎么做的？我看到一个游戏表现是客户端预测+回滚，这就是说，当接收服务器最新的物理状态后，客户端需要对比预测的物理状态，如果不一致的话，需要回滚，然后再次重新计算到当前帧。从网上找到的资料说用的是Phsyx物理引擎，但Phsyx物理引擎是不确定性的，并不能保证回滚再计算能得到与服务器相同的值。这个问题困扰我好久了，希望能得到解答，谢谢。

2021–07–23 回复 喜欢

学就是了

学就是了

推荐阅读

手游完整性校验分析

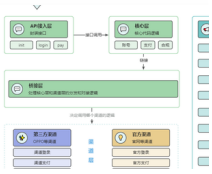
看雪

2017独立游戏发行：代理费比年初翻了五倍 百万元以上也...

佳伦实验室 发表于游戏茶馆归...

手游为什么纷纷出起“PC版”？

触乐 发表于触乐



游戏发行产品系列：SDK

Easy 发表于游戏