KCP协议

1. KCP是什么？

KCP是一个快速可靠协议，能以比 TCP浪费10%-20%的带宽的代价，换取平均延迟降低 30%-40%，且最大延迟降低三倍的传输效果。纯算法实现，并不负责底层协议（如UDP）的收发，需要使用者自己定义下层数据包的发送方式，以 callback的方式提供给 KCP。 连时钟都需要外部传递进来，内部不会有任何一次系统调用。

1. 为什么要用KCP？

传统的通信协议主要有TCP、UDP。

TCP 的优点：可靠，稳定。

TCP 的可靠体现在 TCP 在传递数据之前，会有三次握手来建立连接，而且在数据传递时，有确认、窗口、重传、拥塞控制机制，在数据传完后，还会断开连接用来节约系统资源。

TCP 的缺点：

慢，效率低，占用系统资源高，易被攻击。

TCP 在传递数据之前，要先建连接，这会消耗时间，而且在数据传递时，确认机制、重传机制、拥塞控制机制等都会消耗大量的时间，而且要在每台设备上维护所有的传输连接，事实上，每个连接都会占用系统的 CPU、内存等硬件资源。

而且，因为 TCP 有确认机制、三次握手机制，这些也导致 TCP 容易被人利用，实现 DOS、DDOS、CC 等攻击。

UDP 的优点：

快，比 TCP 稍安全。

UDP 没有 TCP 的握手、确认、窗口、重传、拥塞控制等机制，UDP 是一个无状态的传输协议，所以它在传递数据时非常快。没有 TCP 的这些机制，UDP 较 TCP 被攻击者利用的漏洞就要少一些。但 UDP 也是无法避免攻击的，比如：UDP Flood 攻击。

UDP 的缺点：

不可靠，不稳定。

因为 UDP 没有 TCP 那些可靠的机制，在数据传递时，如果网络质量不好，就会很容易丢包。

TCP 应用场景：

当对网络通讯质量有要求的时候，比如：整个数据要准确无误的传递给对方，这往往用于一些要求可靠的应用，比如 HTTP、HTTPS、FTP 等传输文件的协议，POP、SMTP 等邮件传输的协议。

在日常生活中，常见使用 TCP 协议的应用如下： 浏览器用的 HTTP， FlashFXP 用的 FTP，Outlook 用的 POP、SMTP，Putty 用的 Telnet、SSH，QQ 文件传输。

UDP 应用场景：

当对网络通讯质量要求不高的时候，要求网络通讯速度能尽量的快，这时就可以使用 UDP。

比如，日常生活中，常见使用 UDP 协议的应用如下：QQ 语音，QQ 视频，TFTP 等。有些应用场景对可靠性要求不高会用到 UPD，比如长视频，要求速率。

TCP 与 UDP 区别总结：

1、TCP 面向连接（如打电话要先拨号建立连接）; UDP 是无连接的，即发送数据之前不需要建立连接。

2、TCP 提供可靠的服务。也就是说，通过 TCP 连接传送的数据，无差错，不丢失，不重复，且按序到达；UDP 尽最大努力交付，即不保证可靠交付。

3、TCP 面向字节流，实际上是 TCP 把数据看成一连串无结构的字节流；UDP 是面向报文的。UDP 没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如IP电话，实时视频会议等）

4、每一条 TCP 连接只能是点到点的；UDP支持一对一，一对多，多对一和多对多的交互通信。

对于联机即时战斗游戏而言，首先要保证逻辑运算正确，即所有消息都要按序到达，不能有丢失，而UDP满足不了这一点，TCP可以。其次是低延迟，“玩起来不卡”，客户端能够快速响应玩家的操作，对于这一点，在网络环境好的情况下，TCP也能满足，然而没人可以保证用户的网络环境总是良好的，一旦网络较差，TCP因其确认、窗口、重传等机制的存在，会导致延迟会明显比UDP高出很多。

KCP在一定程度上吸收了两者优点，做到了既快速又可靠地传输，因此可以应用于联机即时战斗的游戏。

1. KCP是怎么实现的？
2. KCP最早是C语言实现。主要接口如下：



ikcp\_create 创建一个 KCP 实例. 传入的 conv 参数标识这个 KCP 连接, 也就是说, 这个连接发出去的每个报文段都会带上 conv, 它也只会接收 conv 与之相等的报文段. 通信的双方必须先协商一对相同的 conv. KCP 本身不提供任何握手机制, 协商 conv 交给使用者自行实现, 比如说通过已有的 TCP 连接协商.在MX1项目中，由Lobby（通过TCP和Client通信）在创建房间时，同时为房间内成员创建KCP conv，并分别发送给Client和 GameServer（KCP和Client通信）。

KCP 是纯算法实现的, 不负责下层协议收发, 内部没有任何系统调用, 连时钟都要外部传进来. 因此需要:

调用 ikcp\_setoutput 设置下层协议输出函数. 每当 KCP 需要发送数据时, 都会回调这个输出函数. 例如下层协议是 UDP 时, 就在输出回调中调用 sendto 将数据发送给对方. 输出回调的 user 参数等于 ikcp\_create 传入的 user 参数.

当下层协议数据到达时, 调用 ikcp\_input 将数据传给 KCP.

以一定的频率调用 ikcp\_update 以驱动 KCP 的时钟. current 表示当前时间, 单位为毫秒.

设置好下层协议和时钟后, 就可以调用 ikcp\_recv 和 ikcp\_send 收发 KCP 数据了.

ikcp\_send 将数据放在发送队列中等待发送;

ikcp\_recv 从接收队列中读取数据;

ikcp\_input 读取下层协议输入数据, 解析报文段; 如果是数据, 就将数据放入接收缓冲区; 如果是 ACK, 就在发送缓冲区中标记对应的报文段为已送达;

ikcp\_flush 调用输出回调将发送缓冲区中的数据发送出去.

2.报文段结构

KCP共有4中报文段

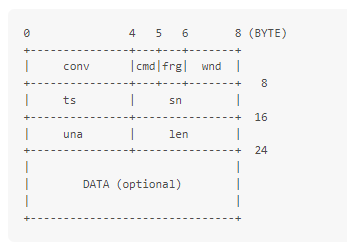
数据报文 IKCP\_CMD\_PUSH

确认报文 IKCP\_CMD\_ACK

窗口探测报文 IKCP\_CMD\_WASK, 询问对端剩余接收窗口的大小.

窗口通知报文 IKCP\_CMD\_WINS, 通知对端剩余接收窗口的大小.

报文段的结构统一都是下图所示



conv 4 字节: 连接标识, 前面已经讨论过了.

cmd 1 字节: Command.

frg 1 字节: 分片数量. 表示随后还有多少个报文属于同一个包.

wnd 2 字节: 发送方剩余接收窗口的大小.

ts 4 字节: 时间戳.

sn 4 字节: 报文编号.

una 4 字节: 发送方的接收缓冲区中最小还未收到的报文段的编号. 也就是说, 编号比它小的报文段都已全部接收.

len 4 字节: 数据段长度.

data: 数据段. 只有数据报文会有这个字段.

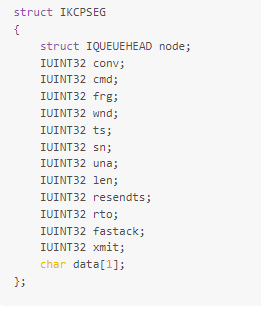
首先, 每个数据报文和 ACK 都会带上 sn, 唯一标识一个报文; 发送方发送一个数据报文, 接收方收到后回一个 ACK, 发送方收到 ACK 后根据 sn 将对应的报文标记为已送达; 同时, 每个报文都会带上 una, 发送方也会根据 una 将相应的报文标记已送达.

ts 可以用来估算 RTT (Round-Trip Time, 往返时间), 从而计算出 RTO (Retransmission TimeOut, 重传超时时间). 会根据 RTO 确定每个报文的超时时间, 如果报文在超时时间内未被标记为已送达, 就会被重传.

数据包的大小可能会超过一个 MSS (Maximum Segment Size, 最大报文段大小). 这个时候需要进行分片, frg 表示随后的分片数量, 即随后还有多少个报文属于同一个包.

每个报文都会带上 wnd, 告诉对端发送方剩余接收窗口的大小, 这有助于对端控制发送速率

3.KCP实例





conv: 连接标识

mtu, mss: 最大传输单元 (Maximum Transmission Unit) 和最大报文段大小. mss = mtu - 包头长度(24).

state: 连接状态, 0 表示连接建立, -1 表示连接断开. (注意 state 是 unsigned int, -1 实际上是 0xffffffff)

snd\_una: 发送缓冲区中最小还未确认送达的报文段的编号. 也就是说, 编号比它小的报文段都已确认送达.

snd\_nxt: 下一个等待发送的报文段的编号.

rcv\_nxt: 下一个等待接收的报文段的编号.

ssthresh: Slow Start Threshold, 慢启动阈值.

rx\_rto: Retransmission TimeOut(RTO), 超时重传时间.

rx\_rttval, rx\_srtt, rx\_minrto: 计算 rx\_rto 的中间变量.

snd\_wnd, rcv\_wnd: 发送窗口和接收窗口的大小.

rmt\_wnd: 对端剩余接收窗口的大小.

cwnd: congestion window, 拥塞窗口. 用于拥塞控制.

probe: 是否要发送控制报文的标志.

current: 当前时间.

interval: flush 的时间粒度.

ts\_flush: 下次需要 flush 的时间.

xmit: 该链接超时重传的总次数.

nrcv\_buf, nsnd\_buf, nrcv\_que, nsnd\_que: 接收缓冲区, 发送缓冲区, 接收队列, 发送队列的长度.

nodelay: 是否启动快速模式. 用于控制 RTO 增长速度.

updated: 是否调用过 ikcp\_update.

ts\_probe, probe\_wait: 确定何时需要发送窗口询问报文.

dead\_link: 当一个报文发送超时次数达到 dead\_link 次时认为连接断开.

incr: 用于计算 cwnd.

snd\_queue, rcv\_queue: 发送队列和接收队列.

snd\_buf, rcv\_buf: 发送缓冲区和接收缓冲区.

acklist, ackcount, ackblock: ACK 列表, ACK 列表的长度和容量. 待发送的 ACK 的相关信息会先存在 ACK 列表中, flush 时一并发送.

buffer: flush 时用到的临时缓冲区.

fastresend: ACK 失序 fastresend 次时触发快速重传.

fastlimit: 传输次数小于 fastlimit 的报文才会执行快速重传.

nocwnd: 是否不考虑拥塞窗口.

stream: 是否开启流模式, 开启后可能会合并包.

logmask: 用于控制日志.

output: 下层协议输出函数.

writelog: 日志函数

4.整体流程

KCP 的整个发送, 接收与重传的流程大体如下:

调用 ikcp\_send 发送数据, 创建报文段实例, 加入 snd\_queue 中.

ikcp\_update 会在合适的时刻调用 ikcp\_flush.

ikcp\_flush 会做:

发送 ACK 列表中所有的 ACK;

检查是否需要发送窗口探测和通知报文, 如果需要就发送相应的报文;

根据发送窗口大小, 将适量的报文段从 snd\_queue 移入到 snd\_buf 中;

将 snd\_buf 中满足条件的报文段都发送出去. 这里的条件有:

新加入 snd\_buf 中, 从未发送过的报文直接发送出去;

发送过的, 但是在 RTO 内未收到 ACK 的报文, 需要重传;

发送过的, 但是 ACK 失序若干次的报文, 需要执行快速重传.

根据丢包情况计算 ssthresh 和 cwnd.

这样, 刚才调用 ikcp\_send 传入的数据就在 ikcp\_flush 中被发送出去了.

报文到达对端.

ikcp\_input 会被调用, 解析收到的数据:

所有的报文都有 una 字段, 根据 una 将相应的报文标记为已送达;

如果是 ACK 报文, 就将相应的报文标记为已送达;

如果是数据报文, 就将它放入 rcv\_buf, 然后将 rcv\_buf 中顺序正确的报文移入 rcv\_queue; 接着将相关信息插入 ACK 列表, 在稍后的 ikcp\_flush 调用中会发送相应的 ACK;

如果是窗口探测报文, 就标记 "需要发送窗口通知". 在稍后的 ikcp\_flush 调用中会发送窗口通知报文;

包括窗口通知报文在内的所有报文都有 wnd 字段, 据此更新 rmt\_wnd;

根据 ACK 失序情况决定快速重传;

计算 cwnd.

调用 ikcp\_recv 接收数据, 从 rcv\_queue 中读取数据.

之后 ACK 会返回给发送方, 发送方在 ikcp\_input 中解析到 ACK, 将对应的报文标记为已送达, 这就完成了一次发送. 如果发生了丢包, 发送方最终收不到 ACK. 这会导致重传.