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，并以 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。

整个协议只有 ikcp.h, ikcp.c两个源文件，可以方便的集成到用户自己的协议栈中。也许你实现了一个P2P，或者某个基于 UDP的协议，而缺乏一套完善的 ARQ可靠协议实现，那么简单的拷贝这两个文件到现有项目中，稍微编写两行代码，即可使用。

URL：<https://github.com/skywind3000/kcp>

**技术特性**

TCP是为流量设计的（每秒内可以传输多少KB的数据），讲究的是充分利用带宽。而KCP是为流速设计的（单个数据包从一端发送到一端需要多少时间），以10%-20%带宽浪费的代价换取了比 TCP快30%-40%的传输速度。TCP信道是一条流速很慢，但每秒流量很大的大运河，而KCP是水流湍急的小激流。KCP有正常模式和快速模式两种，通过以下策略达到提高流速的结果：

* **RTO翻倍vs不翻倍**：TCP超时计算是RTOx2，这样连续丢三次包就变成RTOx8了，十分恐怖，而KCP启动快速模式后不x2，只是x1.5（实验证明1.5这个值相对比较好），提高了传输速度。
* **选择性重传 vs 全部重传**：TCP丢包时会全部重传从丢的那个包开始以后的数据，KCP是选择性重传，只重传真正丢失的数据包。
* **快速重传**：发送端发送了1,2,3,4,5几个包，然后收到远端的ACK: 1, 3, 4, 5，当收到ACK3时，KCP知道2被跳过1次，收到ACK4时，知道2被跳过了2次，此时可以认为2号丢失，不用等超时，直接重传2号包，大大改善了丢包时的传输速度。
* **延迟ACK vs 非延迟ACK** ：TCP为了充分利用带宽，延迟发送ACK（NODELAY都没用），这样超时计算会算出较大RTT时间，延长了丢包时的判断过程。KCP的ACK是否延迟发送可以调节。
* **UNA vs ACK+UNA** ：ARQ模型响应有两种，UNA（此编号前所有包已收到，如TCP）和ACK（该编号包已收到），KCP有单独ACK，且数据包和ACK包都带UNA信息，有效降低ACK丢失成本。
* **非退让流控**：KCP正常模式同TCP一样使用公平退让法则，即发送窗口大小由：发送缓存大小、接收端剩余接收缓存大小、丢包退让及慢启动这四要素决定。但传送及时性要求很高的小数据时，可选择通过配置跳过后两步，仅用前两项来控制发送频率。以牺牲部分公平性及带宽利用率之代价，换取了开着BT都能流畅传输的效果。

**基本使用**

1. 创建 KCP对象：
2. // 初始化 kcp对象，conv为一个表示会话编号的整数，和tcp的 conv一样，通信双方需要
3. // 保证 conv相同，相互的数据包才能够被认可，user是一个给回调函数的指针。

ikcpcb \*kcp = ikcp\_create(conv, user);

1. 设置回调函数：
2. // KCP的下层协议输出函数，KCP需要发送数据时会调用它
3. // buf/len 表示缓存和长度
4. // user指针为 kcp对象创建时传入的值，用于区别多个 KCP对象
5. int udp\_output(const char \*buf, int len, ikcpcb \*kcp, void \*user)
6. {
7. ....
8. }
9. // 设置回调函数

kcp->output = udp\_output;

1. 循环调用 update：
2. // 以一定频率调用 ikcp\_update来更新 kcp状态，并且传入当前的时钟（毫秒单位）。
3. // 比如 10ms调用一次，或用 ikcp\_check确定下次调用 update的时间不必每次调用。

ikcp\_update(kcp, millisec);

1. 输入一个下层数据包：
2. // 收到一个下层数据包（比如UDP包）时需要调用：

ikcp\_input(kcp, received\_udp\_packet, received\_udp\_size);

处理了下层协议的输出/输入后 KCP协议就可以正常工作了，使用 ikcp\_send(kcp, ptr, size)来向远端发送数据。而另一端使用ikcp\_recv(kcp, ptr, size)来接收数据。

**协议配置**

协议默认模式是一个标准的 ARQ，需要通过配置打开各项加速开关：

* **工作模式**

int ikcp\_nodelay(ikcpcb \*kcp, int nodelay, int interval, int resend, int nc);

* + **nodelay** ：是否启用 nodelay模式，0不启用；1启用。
  + **interval** ：协议内部工作的 interval，单位毫秒，比如 10ms或者 20ms
  + **resend** ：快速重传模式，默认0关闭，可以设置2（2次ACK跨越将会直接重传）
  + **nc** ：是否关闭流控，默认是0代表不关闭，1代表关闭。

普通模式：`ikcp\_nodelay(kcp, 0, 40, 0, 0); 极速模式： ikcp\_nodelay(kcp, 1, 10, 2, 1);

* **最大窗口**

int ikcp\_wndsize(ikcpcb \*kcp, int sndwnd, int rcvwnd);

该调用将会设置协议的最大发送窗口和最大接收窗口大小，默认为32.

* **最大传输单元**

纯算法协议并不负责探测 MTU，默认 mtu是1400字节，可以使用ikcp\_setmtu来设置该值。该值将会影响数据包归并及分片时候的最大传输单元。

* **最小RTO**

不管是 TCP还是 KCP计算 RTO时都有最小 RTO的限制，即便计算出来RTO为40ms，由于默认的 RTO是100ms，协议只有在100ms后才能检测到丢包，快速模式下该值为30ms，可以手动更改该值：

kcp->rx\_minrto = 10;

**内存分配器**

默认KCP协议使用 malloc/free进行内存分配释放，如果应用层接管了内存分配，可以用ikcp\_allocator来设置新的内存分配器，注意要在一开始设置：

ikcp\_allocator(my\_new\_malloc, my\_new\_free);

**前向纠错**

为了进一步提高传输速度，下层协议也许会使用前向纠错技术。需要注意，前向纠错会根据冗余信息解出原始数据包。相同的原始数据包不要两次input到KCP，否则将会导致kcp以为对方重发了，这样会产生更多的ack占用额外带宽。

比如下层协议使用最简单的冗余包：单个数据包除了自己外，还会重复存储一次上一个数据包，以及上上一个数据包的内容：

Fn = (Pn, Pn-1, Pn-2)

P0 = (0, X, X)

P1 = (1, 0, X)

P2 = (2, 1, 0)

P3 = (3, 2, 1)

这样几个包发送出去，接收方对于单个原始包都可能被解出3次来（后面两个包任然会重复该包内容），那么这里需要记录一下，一个下层数据包只会input给kcp一次，避免过多重复ack带来的浪费。