

[network \(/tags/#network\)](/tags/#network)[protocol \(/tags/#protocol\)](/tags/#protocol)[TCP/UDP \(/tags/#TCP/UDP\)](/tags/#TCP/UDP)

# KCP: 快速可靠的ARQ协议

*Posted by YuanBao on July 29, 2017*

这段时间看的東西有些雜，先是花了一個星期重新把 golang 的語法回顧了一遍，思考了一下 golang 與 C++ 不同的設計哲學；然後又陸陸續續地看了一些 lock-free 相關的論文以及與之相關的多線程內存模型，總體而言，這些內容在腦海在都還未成體系，因此都先暫時按下不表，今天先花點時間來記錄一個簡單的应用层 ARQ 协议 - KCP (<https://github.com/skywind3000/kcp>)。

## KCP 简介

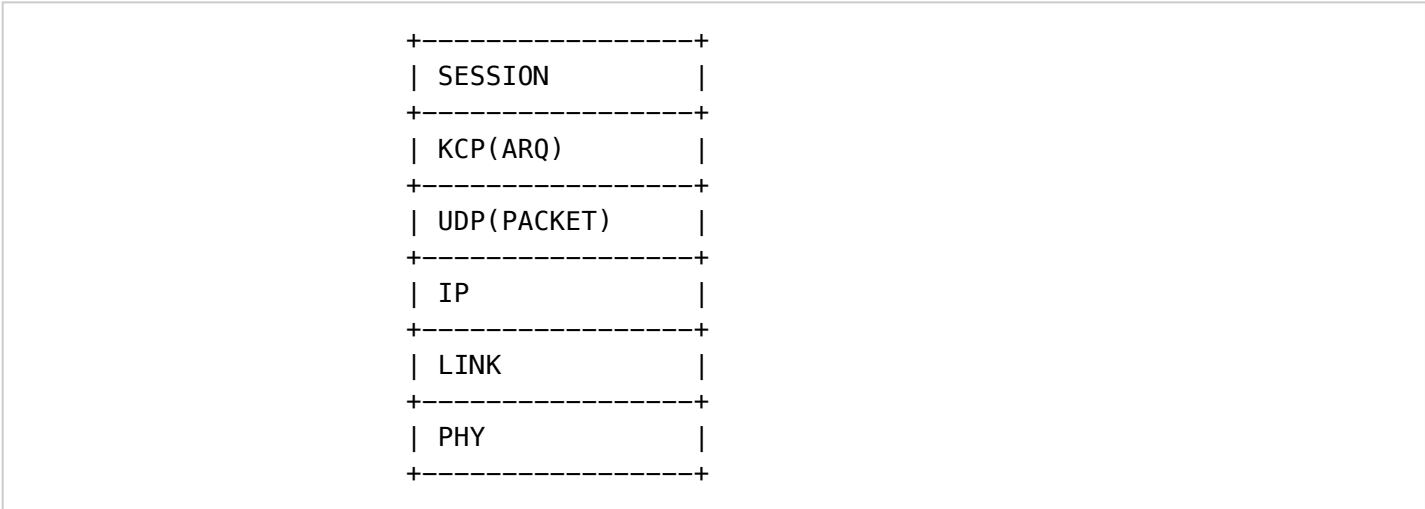
KCP (<https://github.com/skywind3000/kcp>) 是一個快速可靠協議，能以比 TCP 浪費 10%-20% 的帶寬的代價，換取平均延遲降低 30%-40%，且最大延遲降低三倍的傳輸效果。KCP 主要利用了如下思想來加快數據在网络中的傳輸：

1. 相比於 TCP，KCP 啟動快速模式後 超時 RTO 更新不再 x2，而是 x1.5，避免 RTO 快速膨脹。
2. TCP 丟包時會全部重傳從丟的那個包開始以後的數據，KCP 是選擇性重傳，只重傳真正丟失的數據包。
3. TCP 為了充分利用帶寬，延遲發送 ACK（NODELAY 都沒用），這樣超時計算會算出較大 RTT 時間，延長了丟包時的判斷過程。KCP 的 ACK 是否延遲發送可以調節。
4. ARQ 模型響應有兩種，UNA（此編號前所有包已收到，如TCP）和 ACK（該編號包已收到），光用 UNA 將導致全部重傳，光用 ACK 則丟失成本太高，以往協議都是二選其一，而 KCP 協議中，除去單獨的 ACK 包外，所有包都有 UNA 信息。
5. KCP 正常模式同 TCP 一樣使用公平退讓法則，即發送窗口大小由：發送緩存大小、接收端剩餘接收緩存大小、丟包退讓及慢啟動這四要素決定。但傳送及時性要求很高的小數據時，可選擇通過配置跳過後兩步，僅用前兩項來控制發送頻率。

在理论上，以上的几点优化对于一个了解 TCP 协议的程序员都容易理解。在实践上，KCP 已被广泛地应用到游戏（例如 moba 类的王者荣耀）等领域，也证明了其降低传输延迟的有效性。但是从更高的角度而言，**KCP 协议牺牲了网络协议的公平性（TCP Fairness）来贪婪的占用网速，对于提升下一代网络环境而言并不是一个好的方案，其不应该成为 next-net 关注的目标。**相比之下，google 在不久前提出的 BBR (<https://github.com/google/bbr>) 的目标则更加有意义：在下一代中替换 TCP 协议，实现保证传输延迟的前提下最大化地提升网络带宽。关于 BBR 协议的 motivation，以后有时间再慢慢说。

## KCP 实现

简单而言，我们可以把 KCP 协议当做一个应用层协议，这也是为什么 KCP 可以以非侵入式的方式集成到大部分已有的网络传输方案中。下图展示了 KCP 在协议栈中所处的位置（一般而言，KCP 底层均采用 UDP 传输）：



## KCP 基本数据结构

KCP 所使用的 Segment 定义如下，所有不同种类的 KCP 报文均使用相同的报文格式：

```
1 struct IKCPSEG
2 {
3     struct IQUEUEHEAD node;
4     IUINT32 conv, cmd, frg;
5     IUINT32 wnd, ts;
6     IUINT32 sn, una;
7     IUINT32 len;
8     IUINT32 resendts, rto;
9     IUINT32 fastack;
10    IUINT32 xmit;
11    char data[1];
12 };
```

1. node 节点用来串接多个 KCP segment，也就是前向后向指针；
2. conv 是会话编号，通信双方必须一致才能使用 KCP 协议交换数据；
3. cmd 表明当前报文的类型，KCP 共有四种类型：
  - IKCP\_CMD\_PUSH：传输的数据包
  - IKCP\_CMD\_ACK：ACK包，类似于 TCP中的 ACK，通知对方收到了哪些包
  - IKCP\_CMD\_WASK：用来探测远端窗口大小
  - IKCP\_CMD\_WINS：告诉对方自己窗口大小
4. frg 分片的编号，当输出数据大于 MSS 时，需要将数据进行分片，frg 记录了分片时的倒序序号；
5. wnd 填写己方的可用窗口大小，ts 记录了发送时的时间戳，用来估计 RTT；
6. sn 为 data 报文的编号或者 ack 报文的确认编号；
7. una 为当前还未确认的数据包的编号；
8. resendts 为下一次重发该报文的时间，rto 为重传超时时间；
9. fastack 记录了该报文在收到 ACK 时被跳过了几次，用于快重传；
10. xmit 记录了该报文被传输了几次；
11. data 为实际传输的数据 payload；

每一个 KCP 用户都需要调用 `ikcp_create` 创建一个 kcp 控制块 `ikcpcb`。`ikcpcb` 结构用来实现整个 KCP 协议，其成员变量众多，留待后续收发协议过程中介绍。

## KCP 报文发送

由于 KCP 是应用层协议，在使用 KCP 之前，需要先设置底层的输出函数，也就是 `ikcpcb` 中的 `output` 函数，一般而言，KCP 使用者均采用 UDP 作为传输协议。

当设置好输出函数之后，上层应用可以调用 `ikcp_send` 来发送数据。`ikcpcb` 中定义了发送相关的缓冲队列和 `buf`，分别是 `snd_queue` 和 `snd_buf`。应用层调用 `ikcp_send` 后，数据将会进入到 `snd_queue` 中，而下层函数 `ikcp_flush` 将会决定将多少数据从 `snd_queue` 中移到 `snd_buf` 中，进行发送。我们首先来看 `ikcp_send` 的主要功能：

```

1  int ikcp_send(ikcpcb *kcp, const char *buffer, int len)
2  {
3      // 1. 如果当前的 KCP 开启流模式, 取出 `snd_queue` 中的最后一个报文
4      // 将其填充到 mss 的长度, 并设置其 frg 为 0.
5      if (kcp->stream != 0) {
6          if (!iqueue_is_empty(&kcp->snd_queue)) {
7              IKCPSEG *old = iqueue_entry(kcp->snd_queue.prev,
8                                          IKCPSEG, node);
9              ...
10
11         // 2. 计算剩下的数据需要分成几段
12         if (len <= (int)kcp->mss) count = 1;
13         else count = (len + kcp->mss - 1) / kcp->mss;
14         if (count > 255) return -2; // 一次最多发送 255 个报文
15         if (count == 0) count = 1;
16
17         // 3. 为剩下的数据创建 KCP segment
18         for (i = 0; i < count; i++) {
19             int size = len > (int)kcp->mss ? (int)kcp->mss : len;
20             seg = ikcp_segment_new(kcp, size);
21             assert(seg);
22             if (seg == NULL) {
23                 return -2;
24             }
25             if (buffer && len > 0) {
26                 memcpy(seg->data, buffer, size);
27             }
28             seg->len = size;
29             // 流模式情况下分片编号不用填写
30             seg->frg = (kcp->stream == 0)? (count - i - 1) : 0;
31             iqueue_init(&seg->node);
32             iqueue_add_tail(&seg->node, &kcp->snd_queue); // 加入到 snd_queue 中
33             kcp->nsnd_que++;
34             if (buffer) {
35                 buffer += size;
36             }
37             len -= size;
38         }
39     }

```

应用层调用 `ikcp_send` 之后将用户数据置入 `snd_queue` 中, 当 KCP 调用 `ikcp_flush` 时才将数据从 `snd_queue` 中 移入到 `snd_buf` 中, 然后调用 `kcp->output()` 发送。在介绍 `ikcp_flush` 的之前, 我们先看一下 KCP 对于 `ack` 报文的管理。KCP 控制块 `ikcpcb` 中有如下几个成员:

1. `acklist`: 当收到一个数据报文时, 将其对应的 ACK 报文的 `sn` 号以及时间戳 `ts` 同时加入到 `acklist` 中, 即形成如 `[sn1, ts1, sn2, ts2 ...]` 的列表;
2. `ackcount`: 记录 `acklist` 中存放的 ACK 报文的数量;
3. `ackblock`: `acklist` 数组的可用长度, 当 `acklist` 的容量不足时, 需要进行扩容;

接下来看 `ikcp_flush` 的实现, 主要可以分为如下几个部分:

- 检查 `kcp->update` 是否更新, 未更新直接返回。`kcp->update` 由 `ikcp_update` 更新, 上层应用需要每隔一段时间 (10-100ms) 调用 `ikcp_update` 来驱动 KCP 发送数据;
- 准备将 `acklist` 中记录的 ACK 报文发送出去, 即从 `acklist` 中填充 ACK 报文的 `sn` 和 `ts` 字段;
- 检查当前是否需要远端窗口进行探测。由于 KCP 流量控制依赖于远端通知其可接受窗口的大小, 一旦远端接受窗口 `kcp->rmt_wnd` 为0, 那么本地将不会再向远端发送数据, 因此就没有机会从远端接受 ACK 报文, 从而没有机会更新远端窗口大小。在这种情况下, KCP 需要发送窗口探测报文到远端, 待远端回复窗口大小后, 后续传输可以继续:

```
1  if (kcp->rmt_wnd == 0) {
2      if (kcp->probe_wait == 0) { // 初始化探测间隔和下一次探测时间
3          kcp->probe_wait = IKCP_PROBE_INIT;
4          kcp->ts_probe = kcp->current + kcp->probe_wait;
5      }
6      else {
7          if (_itimediff(kcp->current, kcp->ts_probe) >= 0) {
8              if (kcp->probe_wait < IKCP_PROBE_INIT)
9                  kcp->probe_wait = IKCP_PROBE_INIT;
10             kcp->probe_wait += kcp->probe_wait / 2;
11             if (kcp->probe_wait > IKCP_PROBE_LIMIT)
12                 kcp->probe_wait = IKCP_PROBE_LIMIT;
13             kcp->ts_probe = kcp->current + kcp->probe_wait;
14             kcp->probe |= IKCP_ASK_SEND; // 标识需要探测远端窗口
15         }
16     }
17 }
```

- 将窗口探测报文和窗口回复报文发送出去, 这一步用来完成 3 中所说的窗口探测协议;
- 计算本次发送可用的窗口大小, 这里 KCP 采用了可以配置的策略, 正常情况下, KCP 的窗口大小由发送窗口 `snd_wnd`, 远端接收窗口 `rmt_wnd` 以及根据流控计算得到的 `kcp->cwnd` 共同决定; 但是当开启了 `nocwnd` 模式时, 窗口大小仅由前两者决定;

- 将缓存在 `snd_queue` 中的数据移到 `snd_buf` 中等待发送，这个两个 `buf` 的作用在前文中已经介绍；
- 在发送数据之前，先设置快重传的次數和重传间隔；KCP 允许设置快重传的次數，即 `fastresend` 参数。例如设置 `fastresend` 为2，并且发送端发送了1,2,3,4,5几个包，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；每个报文的 `fastack` 记录了该报文被跳过了几次，由函数 `ikcp_parse_fastack` 更新。于此同时，KCP 也允许设置 `nodelay` 参数，当激活该参数时，每个报文的超时重传时间将由  $x2$  变为  $x1.5$ ，即加快报文重传：

```

1 // 是否设置了快重传次数
2 resent = (kcp->fastresend > 0)? (IUINT32)kcp->fastresend : 0xffffffff;
3 // 是否开启了 nodelay
4 rtomin = (kcp->nodelay == 0)? (kcp->rx_rto >> 3) : 0;
5

```

- 将 `snd_buf` 中的数据发送出去，这里分为几种不同的情况处理：

```

1 if(segment->xmit == 0) {
2     // 1. 如果该报文是第一次传输，那么直接发送
3 }
4 else if (_itimediff(current, segment->resendts) >= 0) {
5     // 2. 如果已经到了该报文的重新时间，那么发送该报文
6     if (kcp->nodelay == 0) { // 根据 nodelay 参数更新重传时间
7         segment->rto += kcp->rx_rto;
8     } else {
9         segment->rto += kcp->rx_rto / 2;
10    }
11    segment->resendts = current + segment->rto;
12    lost = 1; // 记录出现了报文丢失
13 }
14 else if (segment->fastack >= resent) {
15     // 3. 如果该报文被跳过的次数超过了设置的快重传次数，发送该报文
16     segment->fastack = 0;
17     segment->resendts = current + segment->rto;
18     change++; // 标识快重传发生
19 }

```

- 根据设置的 `lost` 和 `change` 更新窗口大小；注意 快重传和丢包时的窗口更新算法不一致，这一点类似于 TCP 协议的拥塞控制和快恢复算法；

KCP 的报文发送流程到此已经分析完了，整个过程很容易理解，接下来我们结合上面的分析来看报文接收的流程。

## KCP 报文接收

对应于 `ikcp_send` 的应用层接收函数为 `ikcp_recv`，其主要执行的流程如下：

- 首先检测一下本次接收数据之后，是否需要窗口恢复。在前面的内容中解释过，KCP 协议在远端窗口为0的时候将会停止发送数据，此时如果远端调用 `ikcp_recv` 将数据从 `rcv_queue` 中移动到应用层 `buffer` 中之后，表明其可以再次接受数据，为了能够恢复数据的发送，远端可以主动发送 `IKCP_ASK_TELL` 来告知窗口大小；

```
1  if (kcp->nrcv_que >= kcp->rcv_wnd)
2      recover = 1;  // 标记可以开始窗口恢复
```

- 开始将 `rcv_queue` 中的数据根据分片编号 `frg` merge 起来，然后拷贝到用户的 `buffer` 中。这里 `ikcp_recv` 循环遍历 `rcv_queue`，按序拷贝数据，当碰到某个 `segment` 的 `frg` 为 0 时跳出循环，表明本次数据接收结束。这点应该很好理解，经过 `ikcp_send` 发送的数据会进行分片，分片编号为倒序序号，因此 `frg` 为 0 的数据包标记着完整接收到了一次 `send` 发送过来的数据；

```
1  for (len = 0, p = kcp->rcv_queue.next; p != &kcp->rcv_queue; ) {
2      int fragment;
3      seg = iqueue_entry(p, IKCPSEG, node);
4      p = p->next;
5
6      if (buffer) {
7          memcpy(buffer, seg->data, seg->len);
8          buffer += seg->len;
9      }
10
11     len += seg->len;
12     fragment = seg->frg;
13
14     ...
15     if (fragment == 0)
16         break;
17 }
```



- 下一步将 `rcv_buf` 中的数据转移到 `rcv_queue` 中，这个过程根据报文的 `sn` 编号来确保转移到 `rcv_queue` 中的数据一定是按序的：

```

1  while (! iqueue_is_empty(&kcp->rcv_buf)) {
2      IKCPSEG *seg = iqueue_entry(kcp->rcv_buf.next, IKCPSEG, node);
3      // 1. 根据 sn 确保数据是按序转移到 rcv_queue 中
4      // 2. 根据接收窗口大小来判断是否可以接收数据
5      if (seg->sn == kcp->rcv_nxt && kcp->nrcv_que < kcp->rcv_wnd) {
6          iqueue_del(&seg->node);
7          kcp->nrcv_buf--;
8          iqueue_add_tail(&seg->node, &kcp->rcv_queue);
9          kcp->nrcv_que++;
10         kcp->rcv_nxt++;
11     } else {
12         break;
13     }
14 }

```

- 最后进行窗口恢复。此时如果 `recover` 标记为1，表明在此次接收之前，可用接收窗口为0，如果经过本次接收之后，可用窗口大于0，将主动发送 `IKCP_ASK_TELL` 数据包来通知对方已可以接收数据：

```

1  if (kcp->nrcv_que < kcp->rcv_wnd && recover) {
2      kcp->probe |= IKCP_ASK_TELL; // 将会在 ikcp_flush 中发送
3  }

```

`ikcp_rcv` 仅为上层调用的接口，KCP 协议需要从底层接受数据到 `rcv_buf` 中，这是通过函数 `ikcp_input` 实现。`ikcp_input` 中的所有功能都在一个外层的循环中实现：

- 首先将接收到的数据包进行解码，并进行基本的数据包长度和类型校验；KCP 协议只会接收到前文中所介绍的四种数据包；
- 调用 `ikcp_parse_una` 来确定已经发送的数据包有哪些被对方接收到。注意 KCP 中所有的报文类型均带有 `una` 信息。前面介绍过，发送端发送的数据都会缓存在 `snd_buf` 中，直到接收到对方确认信息之后才会删除。当接收到 `una` 信息后，表明 `sn` 小于 `una` 的数据包都已经被对方接收到，因此可以直接从 `snd_buf` 中删除。同时调用 `ikcp_shrink_buf` 来更新 KCP 控制块的 `snd_una` 数值。
- 处理 `IKCP_CMD_ACK` 报文：

1. 调用 `ikcp_update_ack` 来根据 ACK 时间戳更新本地的 `rtt`，这类似于 TCP 协议；
2. 之后调用函数 `ikcp_parse_ack` 来根据 ACK 的编号确认对方收到了哪个数据包；**注意KCP 中同时使用了 UNA 以及 ACK 编号的报文确认手段。UNA 表示此前所有的数据都已经被接收到，而 ACK 表示指定编号的数据包被接收到；**
3. 调用 `ikcp_shrink_buf` 来更新 KCP 控制块的 `snd_una`；
4. 记录当前收到的最大的 ACK 编号，在快重传的过程计算已发送的数据包被跳过的次数；

```

1  if (cmd == IKCP_CMD_ACK) {
2      if (_itimediff(kcp->current, ts) >= 0) {
3          ikcp_update_ack(kcp, _itimediff(kcp->current, ts));
4      }
5      ikcp_parse_ack(kcp, sn); // 更新 rtt
6      ikcp_shrink_buf(kcp);    // 更新控制块的 snd_una
7      if (flag == 0) {
8          flag = 1;
9          maxack = sn;          // 记录最大的 ACK 编号
10     } else {
11         if (_itimediff(sn, maxack) > 0) {
12             maxack = sn;      // 记录最大的 ACK 编号
13         }
14     }
15 }

```

- 处理 `IKCP_CMD_PUSH` 报文：

1. 对于来自于对方的标准数据包，首先需要检测该报文的编号 `sn` 是否在窗口范围内；
2. 调用 `ikcp_ack_push` 将对该报文的确认 ACK 报文放入 ACK 列表中，ACK 列表的组织方式在前文中已经介绍；
3. 最后调用 `ikcp_parse_data` 将该报文插入到 `rcv_buf` 链表中；

- 对于接收到的 `IKCP_CMD_WASK` 报文，直接标记下次将发送窗口通知报文；而对于报文 `IKCP_CMD_WINS` 无需做任何特殊操作；
- 根据记录的最大的 ACK 编号 `maxack` 来更新 `snd_buf` 中的报文的 `fastack`，这个过程在介绍 `ikcp_flush` 中提到过，对于 `fastack` 大于设置的 `resend` 参数时，将立马进行快重传；
- 最后，根据接收到报文的 `una` 和 KCP 控制块的 `una` 参数进行流控；

到此为止有关 KCP 整个协议的发送和接收逻辑都介绍完了。当然，使用 KCP 时还有两个关键的函数 `ikcp_update` 和 `ikcp_checkout`，这两个函数在了解发送和接收流程之后容易理解，这里不在赘述了。后续如果有时间，再来讲一讲 google 的 BBR 协议，相比于 KCP，个人觉得 BBR 的意义似乎更加深

远，设计也更加科学。


**PREVIOUS**

LEVELDB 笔记六：MEMTABLE 实现  
(/2017/07/20/LEVELDB-06/)

**NEXT**

MEMORY REORDERING 浅析  
(/2017/09/22/MEMORY-BARRIER/)

3条评论 A coder is also a poet  Disqus 隐私政策

 登录 ▾

 Favorite 2

 推文

 分享

评分最高 ▾



加入讨论...

通过以下方式登录

或注册一个 DISQUS 帐号 

姓名



**John Wu** • 4 年前

请教下，怎么理解kcp说的，浪费带宽换取延迟降低的说法，体现在哪些地方，谢谢

1 ^ | ▾ 1 • 回复 • 分享 ▸



**Scofield Micheal** → John Wu • 3 年前

体现在重试的机制里

^ | ▾ • 回复 • 分享 ▸



**Scofield Micheal** • 3 年前

看了好多kcp的分析，这篇手动点赞

^ | ▾ • 回复 • 分享 ▸

 订阅  在您的网站上使用 Disqus添加 Disqus添加  不要出售我的数据

**FEATURED TAGS (/tags/)**

life (/tags/#life)

protocol (/tags/#protocol)

programming (/tags/#programming)

network (/tags/#network)

C++ (/tags/#C++)

python (/tags/#python)

OpenGL (/tags/#OpenGL)

NoSQL (/tags/#NoSQL)

leveldb (/tags/#leveldb)

libco (/tags/#libco)


coroutine (/tags/#coroutine)

lock-free (/tags/#lock-free)

Distributed (/tags/#Distributed)

## FRIENDS

Lily BBS (<http://bbs.nju.edu.cn/>) Sina Blog (<http://weibo.com/littleyuanbao>)

 (</feed.xml>)

 (<https://www.zhihu.com/people/yuan-bao-87>)

 (<http://weibo.com/littleyuanbao>)

 (<https://github.com/kaiywen>)

Copyright © A Coder is a Poet 2019

Theme by Hux (<http://huangxuan.me>) | [Star](#)