讲师介绍--专业来自专注和实力

Darren老师

曾供职于国内知名半导体公司（珠海扬智/深圳联发科），曾在某互联网公司担任音视频通话项目经理。主要从事音视频驱动、多媒体中间件、流媒体服务器的开发，开发过即时通讯+音视频通话的大型项目，在音视频、C/C++/GO Linux服务器领域有丰富的实战经验。

课程安排

1. 如果做到可靠性传输
2. UDP与TCP，我们如何选择
3. UDP如何可靠，KCP协议在哪些方面有优势
4. KCP协议精讲（重点讲解）
5. QUIC时代是否已经到来

1 如何做到可靠性传输

- ACK机制
- 重传机制 重传策略
- 序号机制　　3 2 1 -> 2 3 1
- 重排机制　　2 3 1 -> 3 2 1
- 窗口机制

2.1 UDP与TCP，我们如何选择

选项	UDP	TCP
是否连接	无连接	面向连接
是否可靠	不可靠传输，不使用流量控制和拥塞控制	可靠传输，使用流量控制和拥塞控制
连接对象个数	支持一对一，一对多，多对一和多对多交互通信	只能是一对一通信
传输方式	面向报文	面向字节流
首部开销	首部开销小，仅8字节	首部最小20字节，最大60字节
适用场景	适用于实时应用（IP电话、视频会议、直播等） 游戏行业、物联网行业	适用于要求可靠传输的应用，例如文件传输

2.2 TCP和UDP格式对比

16 位源端口号			16 位目标端口号
32 位序列号			
32 位确认号			
4 位头长	4 位保留	8 位标志位	16 位窗口大小
16 位校验和			16 位紧急指针
选项			
数据			

TCP格式

16 位源端口号	16 位目标端口号
16 位长度	16 位校验和
数据	

UDP格式

在网络中，我们认为传输是不可靠的，而在很多场景下我们需要的是可靠的数据，所谓的可靠，指的是数据能够正常收到，且能够顺序收到，于是就有了ARQ协议，TCP之所以可靠就是基于此。

2.3 ARQ协议(Automatic Repeat-reQuest)

ARQ协议(Automatic Repeat-reQuest)，即自动重传请求，是传输层的错误纠正协议之一，它通过使用确认和超时两个机制，在不可靠的网络上实现可靠的信息传输。

ARQ协议主要有3种模式：

1. 即停等式(stop-and-wait)ARQ
2. 回退n帧(go-back-n)ARQ,
3. 选择性重传(selective repeat)ARQ

2.3.1 ARQ协议-停等式(stop-and-wait)

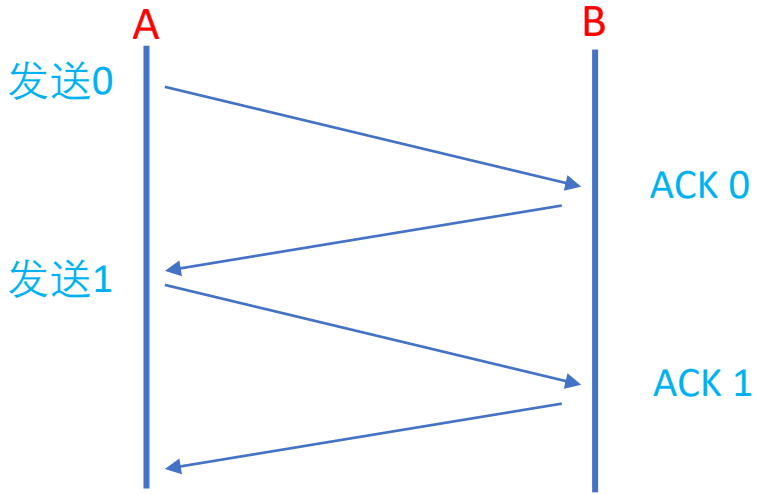
停等协议的工作原理如下:

1、发送方对接收方发送数据包，然后等待接收方回复ACK并且开始计时。

2、在等待过程中，发送方停止发送新的数据包。

3、当数据包没有成功被接收方接收，接收方不会发送ACK.这样发送方在等待一定时间后，重新发送数据包。

4、反复以上步骤直到收到从接收方发送的ACK.



缺点：较长的等待时间导致低的数据传输速度。

2.3.2 ARQ协议-回退n帧(go-back-n)ARQ 1

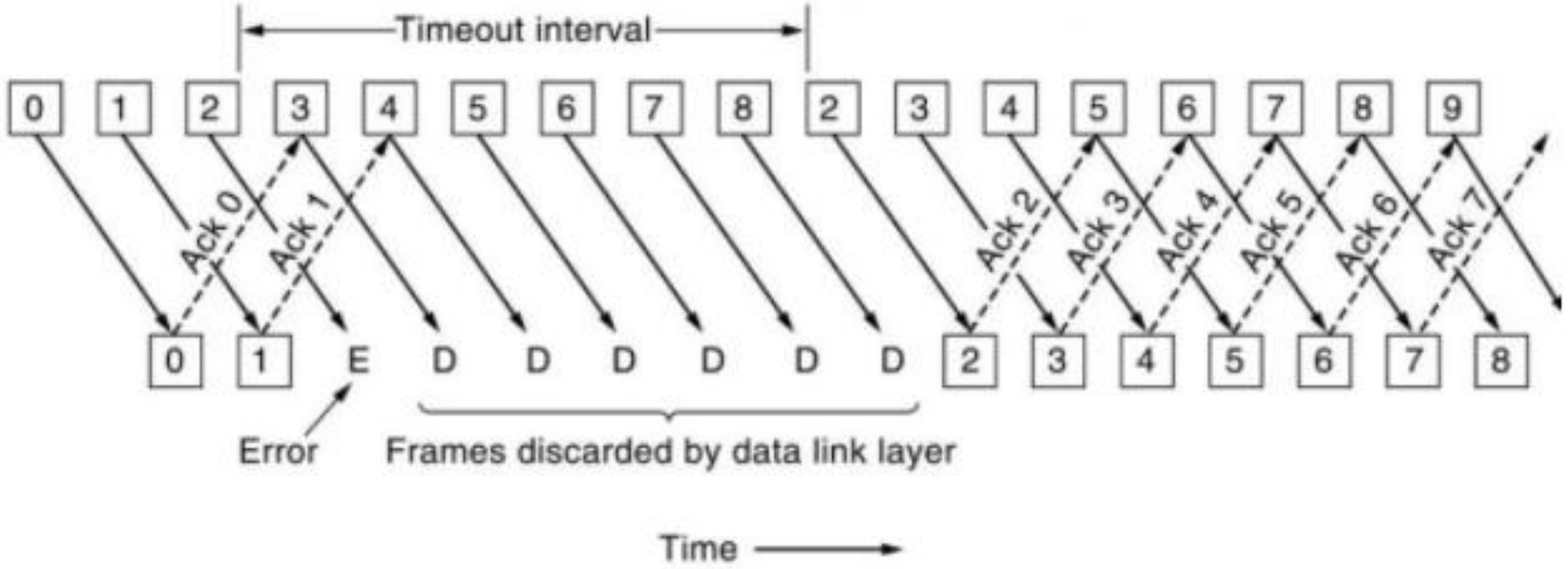
为了克服停等协议长时间等待ACK的缺陷，连续ARQ协议会连续发送一组数据包，然后再等待这些数据包的ACK。

什么是滑动窗口：发送方和接收方都会维护一个数据帧的序列，这个序列被称作窗口。发送方的窗口大小由接收方确定，目的在于控制发送速度，以免接收方的缓存不够大，而导致溢出，同时控制流量也可以避免网络拥塞。协议中规定，对于窗口内未经确认的分组需要重传。

回退N步(Go-Back-N,GBN)：回退N步协议允许发送方在等待超时的间歇，可以继续发送分组。所有发送的分组，都带有序号。在GBN协议中，发送方需响应以下三种事件：
1、上层的调用。上层调用相应send()时，发送方首先要检查发送窗口是否已满。
2、接收ACK。在该协议中，对序号为n的分组的确认采取累积确认的方式，表明接收方已正确接收到序号n以前(包括n)的所有分组。
3、超时。若出现超时，发送方将重传所有已发出但还未被确认的分组

2.3.2 ARQ协议-回退n帧(go-back-n)ARQ 2

对于接收方来说，若一个序号为n的分组被正确接收，并且按序，则接收方会为该分组返回一个ACK给发送方，并将该分组中的数据交付给上层。在其他情况下，接收方都会丢弃分组。若分组n已接收并交付，那么所有序号比n小的分组也已完成了交付。因此GBN采用累积确认是一个很自然的选择。发送方在发完一个窗口里的所有分组后，会检查最大的有效确认，然后以最大有效确认的后一个分组开始重传



如上图所示，序号为2的分组丢失，因此**分组2及之后的分组都将被重传**。

总结：GBN采用的技术包括序号、累积确认、检验和以及计时/重传。

2.3.3 ARQ协议-选择重传(Selective-repeat) 1

虽然GBN改善了停等协议中时间等待较长的缺陷，但它依旧存在着性能问题。特别是当窗口长度很大的时候，会使效率大大降低。而SR协议通过让发送方仅重传在接收方丢失或损坏了的分组，从而避免了不必要的重传，提高了效率。

在SR协议下，发送方需响应以下三种事件：

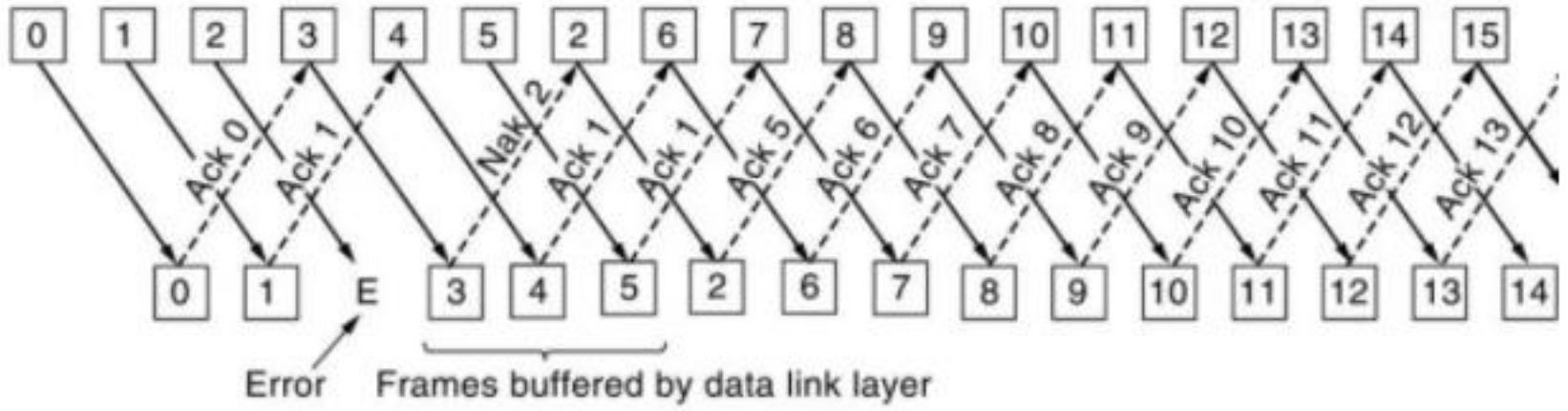
1、从上层收到数据。当从上层收到数据后，发送方需检查下一个可用于该分组的序号。若序号在窗口中则将数据发送。

2、接收ACK。若收到ACK，且该分组在窗口内，则发送方将那个被确认的分组标记为已接收。若该分组序号等于基序号，则窗口序号向前移动到具有最小序号的未确认分组处。若窗口移动后并且有序号落在窗口内的未发送分组，则发送这些分组。

3、超时。若出现超时，发送方将重传已发出但还未确认的分组。与GBN不同的是，SR协议中的每个分组都有独立的计时器。

2.3.3 ARQ协议-选择重传(Selective-repeat) 2

在SR协议下，接收方需响应以下三种事件:
(假设接收窗口的基序号为4，分组长度也为4)
1、序号在[4,7]内的分组被正确接收。该情况下，收到的分组落在接收方的窗口内，一个ACK将发送给发送方。若该分组是以前没收到的分组，则被缓存。若该分组的序号等于基序号4，则该分组以及以前缓存的序号连续的分组都交付给上层，然后，接收窗口将向前移动。
2、序号在[0,3]内的分组被正确接收。在该情况下，必须产生一个ACK，尽管该分组是接收方以前已确认过的分组。若接收方不确认该分组，发送方窗口将不能向前移动。
3、其他情况。忽略该分组
对于接收方来说，若一个分组正确接收而不管其是否按序，则接收方会为该分组返回一个ACK给发送方。失序的分组将被缓存，直到所有丢失的分组都被收到，这时才可以将一批分组按序交付给上层。



2.4 RTT和RTO

- RTO（Retransmission TimeOut）即重传超时时间。

- RTT(Round-Trip Time)： 往返时延。表示从发送端发送数据开始，到发送端收到来自接收端的确认（接收端收到数据后便立即发送确认），总共经历的时延。
  由三部分组成：
  - 链路的传播时间（propagation delay）
  - 末端系统的处理时间、
  - 路由器缓存中的排队和处理时间（queuing delay）

  其中，前两个部分的值对于一个TCP连接相对固定，路由器缓存中的排队和处理时间会随着整个网络拥塞程度
  的变化而变化。所以RTT的变化在一定程度上反应了网络的拥塞程度。

进一步阅读参考：TCP中RTT的测量和RTO的计算
https://blog.csdn.net/zhangskd/article/details/7196707

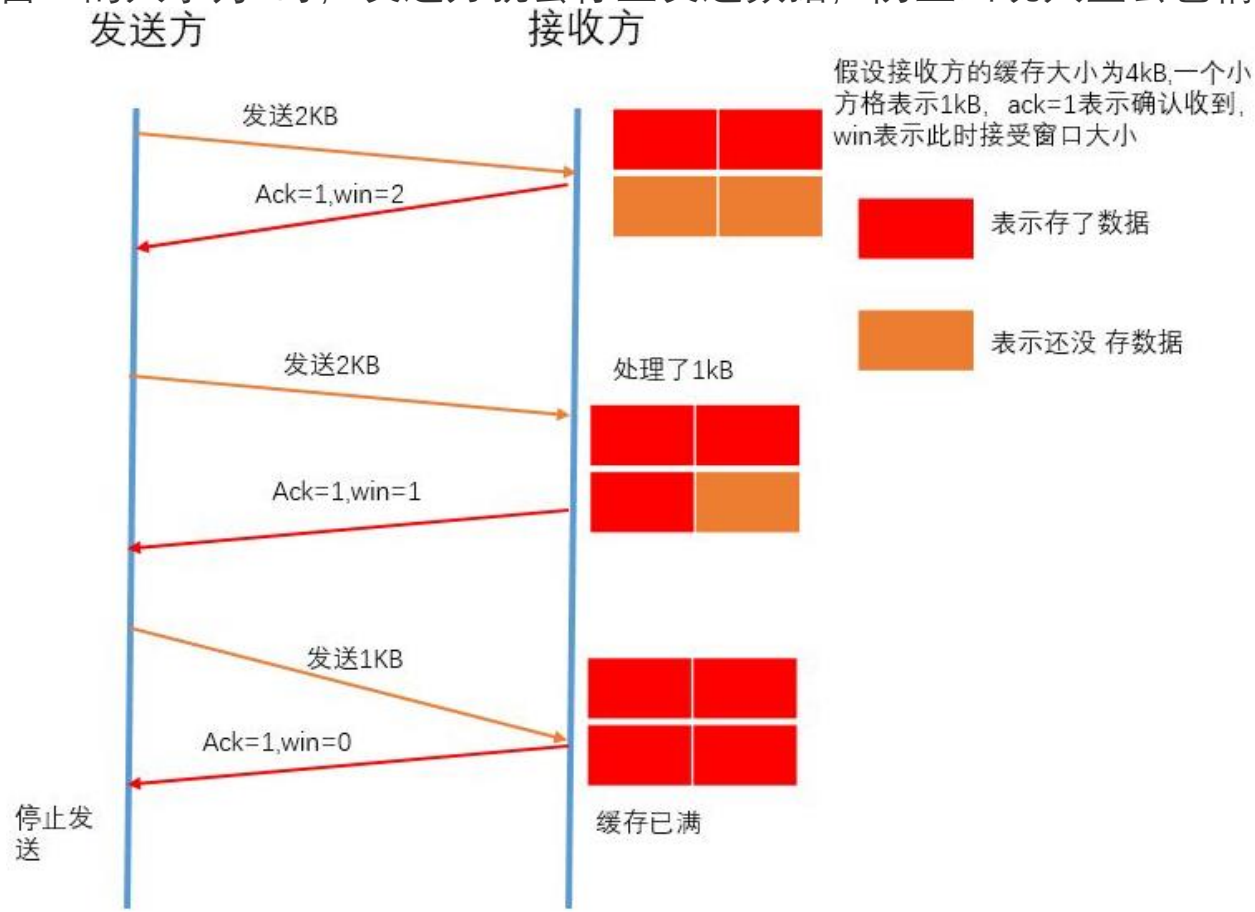
在TCP的选项 内容可以插入时间戳。

2.4 流量控制

- 双方在通信的时候，发送方的速率与接收方的速率是不一定相等，如果发送方的发送速率太快，会导致接收方处理不过来，这时候接收方只能把处理不过来的数据存在缓存区里（失序的数据包也会被存放在缓存区里）。
- 如果缓存区满了发送方还在疯狂着发送数据，接收方只能把收到的数据包丢掉，大量的丢包会极大着浪费网络资源，因此，我们需要控制发送方的发送速率，让接收方与发送方处于一种动态平衡才好。
- **对发送方发送速率的控制，称之为流量控制。**

2.4 流量控制-如何控制？

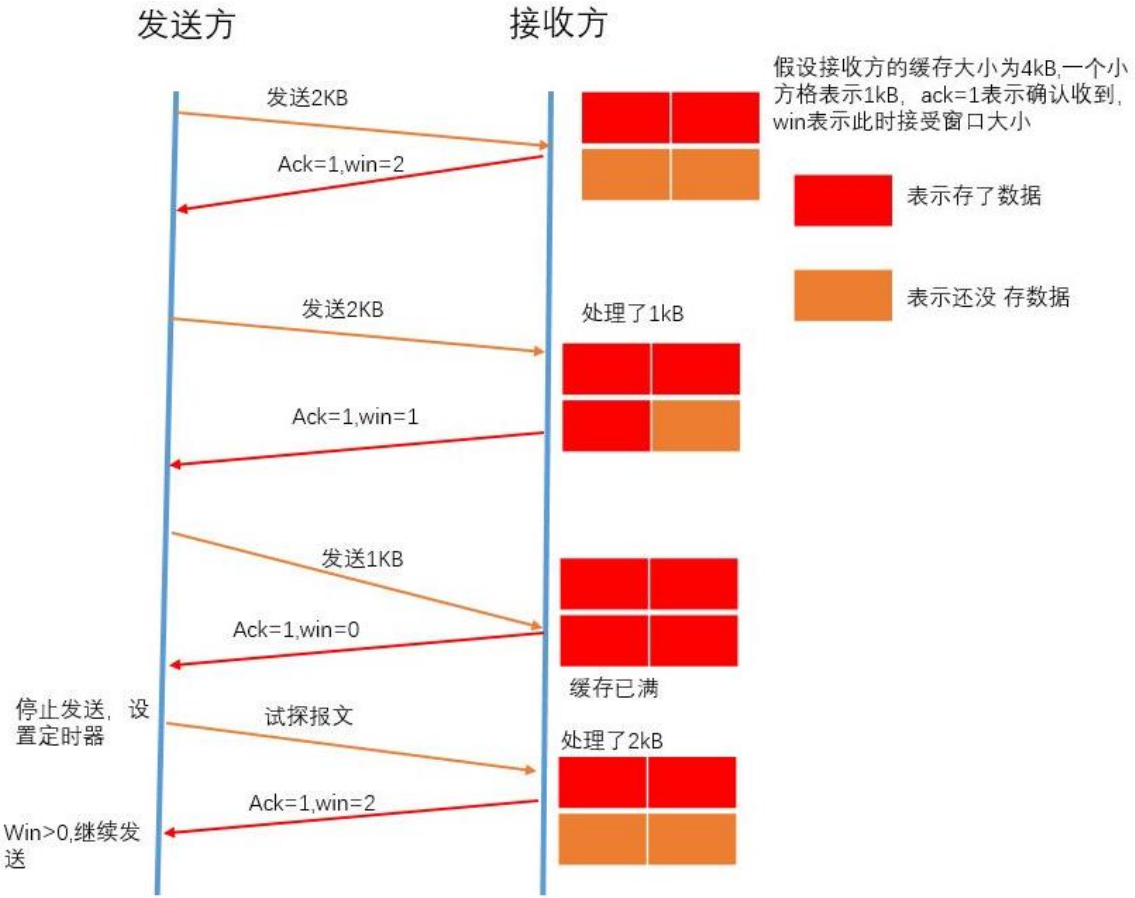
- 接收方每次收到数据包，可以在发送确定报文的时候，同时告诉发送方自己的缓存区还剩余多少是空闲的，我们也把缓存区的剩余大小称之为接收窗口大小，用变量win来表示接收窗口的大小。
- 发送方收到之后，便会调整自己的发送速率，也就是调整自己发送窗口的大小，当发送方收到接收窗口的大小为0时，发送方就会停止发送数据，防止出现大量丢包情况的发生。



2.4 流量控制-发送方何时再继续发送数据?

当发送方停止发送数据后，该怎样才能知道自己可以继续发送数据?

1. 当接收方处理好数据，接受窗口 win > 0 时，**接收方发个通知报文去通知发送方**，告诉他可以继续发送数据了。当发送方收到窗口大于0的报文时，就继续发送数据。
2. 当发送方收到接受窗口 win = 0 时，这时发送方停止发送报文，并且同时开启一个定时器，**每隔一段时间就发个测试报文去询问接收方**，打听是否可以继续发送数据了，如果可以，接收方就告诉他此时接受窗口的大小；如果接受窗口大小还是为0，则发送方再次刷新启动定时器。



2.4 流量控制-小结

1. 通信的双方都拥有两个滑动窗口，一个用于接受数据，称之为接收窗口；一个用于发送数据，称之为拥塞窗口(即发送窗口)。指出接受窗口大小的通知我们称之为窗口通告。
2. **接收窗口的大小固定吗？**接受窗口的大小是根据某种算法动态调整的。
3. **接收窗口越大越好吗？**当接收窗口达到某个值的时候，再增大的话也不怎么会减少丢包率的了，而且还会更加消耗内存。所以接收窗口的大小必须根据网络环境以及发送发的的拥塞窗口来动态调整。
4. **发送窗口和接受窗口相等吗？**接收方在发送确认报文的时候，会告诉发送发自己的接收窗口大小，而发送方的发送窗口会据此来设置自己的发送窗口，但这并不意味着他们就会相等。首先接收方把确认报文发出去的那一刻，就已经在一边处理堆在自己缓存区的数据了，所以一般情况下接收窗口 >= 发送窗口。

2. 5拥塞控制

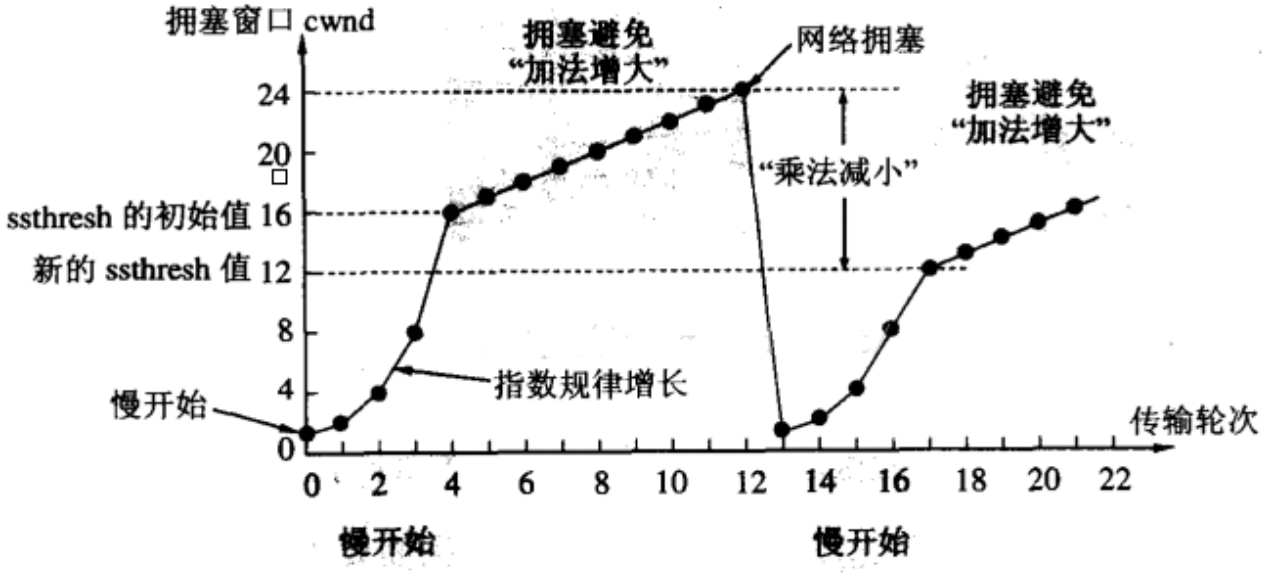


图 5-25　慢开始和拥塞避免算法的实现举例

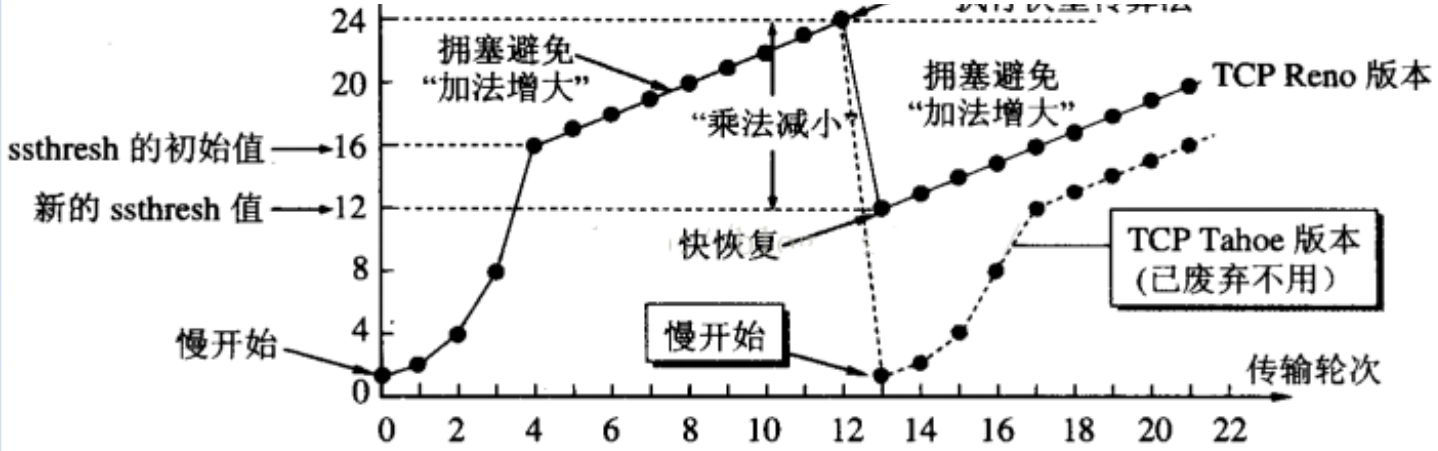


图 5-27　从连续收到三个重复的确认转入拥塞避免

塞控制和流量控制虽然采取的动很相似，但拥塞控制与网络的拥情况相关联，而流量控制与接收的缓存状态相关联。

多阅读参考：5分钟读懂拥塞制 (qq.com)

柚子老师: 2690491738

3 UDP如何可靠，KCP协议在哪些方面有优势

以10%-20%带宽浪费的代价换取了比 TCP快30%-40%的传输速度。

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。 200 300 450 675 – 200 400 800 1600

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（跳过多少个包马上重传）（如果使用了快速重传，可以不考虑RTO））：

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。 fastresend =2

3 UDP如何可靠，KCP协议在哪些方面有优势2

以10%-20%带宽浪费的代价换取了比 TCP快30%-40%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），光用UNA将导致全部重传，光用ACK则丢失成本太高，以往协议都是二选其一，而 **KCP**协议中，除去单独的 ACK包外，所有包都有UNA信息。

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。

4 KCP精讲-名词说明

- kcp官方：https://github.com/skywind3000/kcp

- 名词说明

用户数据：应用层发送的数据，如一张图片2Kb的数据

MTU：最大传输单元。即每次发送的最大数据，1500 **实际使用1400**

RTO：Retransmission TimeOut，重传超时时间。

cwnd:congestion window，拥塞窗口，表示发送方可发送多少个KCP数据包。与接收方窗口有关，与网络状况（拥塞控制）有关，与发送窗口大小有关。

rwnd:receiver window,接收方窗口大小，表示接收方还可接收多少个KCP数据包

snd_queue:待发送KCP数据包队列

snd_nxt:下一个即将发送的kcp数据包序列号

snd_una:下一个待确认的序列号

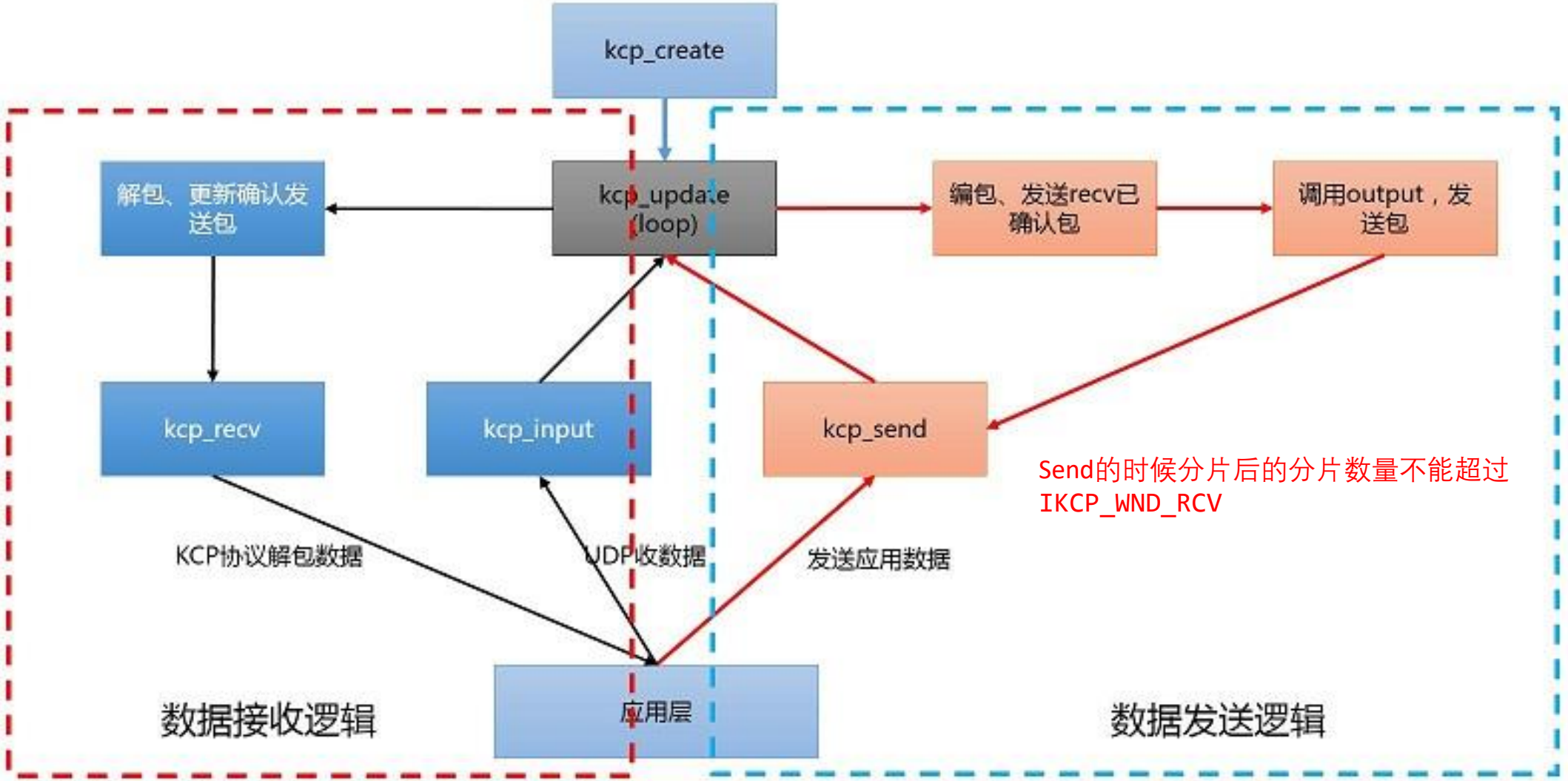
4. 2 kcp使用方式

1. 创建 KCP对象：ikcpcb *kcp = ikcp_create(conv, user);
2. 设置传输回调函数（如UDP的send函数）： kcp->output = udp_output;
    1. 真正发送数据需要调用sendto
3. 循环调用 update：ikcp_update(kcp, millisec);
4. 输入一个应用层数据包（如UDP收到的数据包）：
   ikcp_input(kcp,received_udp_packet,received_udp_size);
    1. 我们要使用recvfrom接收，然后扔到kcp里面做解析
5. 发送数据：ikcp_send(kcp1, buffer, 8);　　用户层接口
6. 接收数据：hr = ikcp_recv(kcp2, buffer, 10);

问题

- sendto每次发送多长的数据?
- ikcp_send可以发送多大长度的数据?
- 如何进行ack?
- 窗口机制如何实现?

4.3 kcp源码流程图


kcp_create
解包、更新确认发送包
kcp_update (loop)
编包、发送recv已确认包
调用output，发送包
kcp_recv
kcp_input
kcp_send
Send的时候分片后的分片数量不能超过IKCP_WND_RCV
KCP协议解包数据
UDP收数据
发送应用数据
应用层
数据接收逻辑
数据发送逻辑

4.4 kcp配置模式

1. 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。

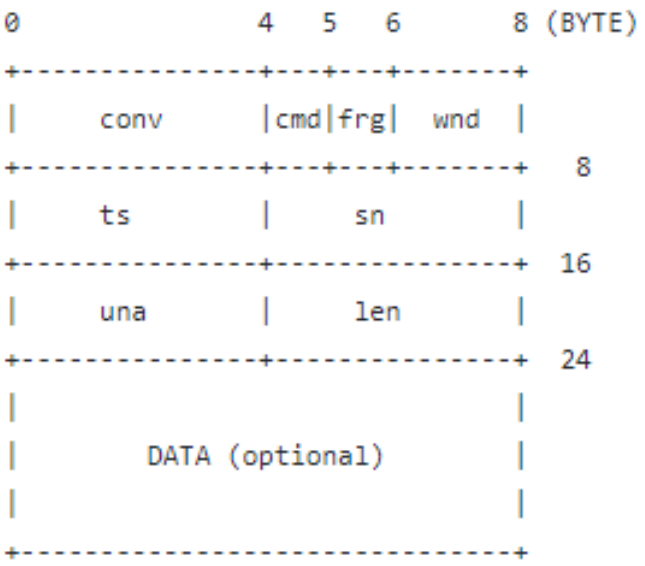
   普通模式： ikcp_nodelay(kcp, 0, 40, 0, 0);
   极速模式： ikcp_nodelay(kcp, 1, 10, 2, 1)

2. 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，单位为包。

3. 最大传输单元：int ikcp_setmtu(ikcpcb *kcp, int mtu);
   kcp协议并不负责探测 MTU，默认 mtu是1400字节

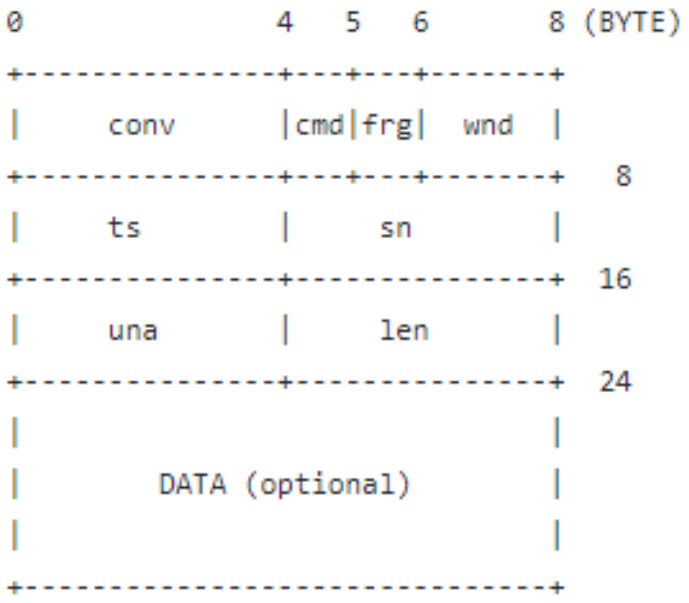
4. 最小RTO：不管是 TCP还是 KCP计算 RTO时都有最小 RTO的限制，即便计算出来RTO为40ms，由于默认的 RTO是100ms，协议只有在100ms后才能检测到丢包，快速模式下为30ms，可以手动更改该值： kcp->rx_minrto = 10;

4.5 kcp协议头



- [0,3]conv:连接号。UDP是无连接的，conv用于表示来自于哪个客户端。对连接的一种替代
- [4]cmd:命令字。如，IKCP_CMD_ACK确认命令，IKCP_CMD_WASK接收窗口大小询问命令，IKCP_CMD_WINS接收窗口大小告知命令，
- [5]frg:分片，用户数据可能会被分成多个KCP包，发送出去
- [6,7]wnd:接收窗口大小，发送方的发送窗口不能超过接收方给出的数值
- [8,11]ts:时间序列
- [12,15]sn:序列号
- [16,19]una:下一个可接收的序列号。其实就是确认号，收到sn=10的包，una为11
- [20,23]len：数据长度
- data:用户数据

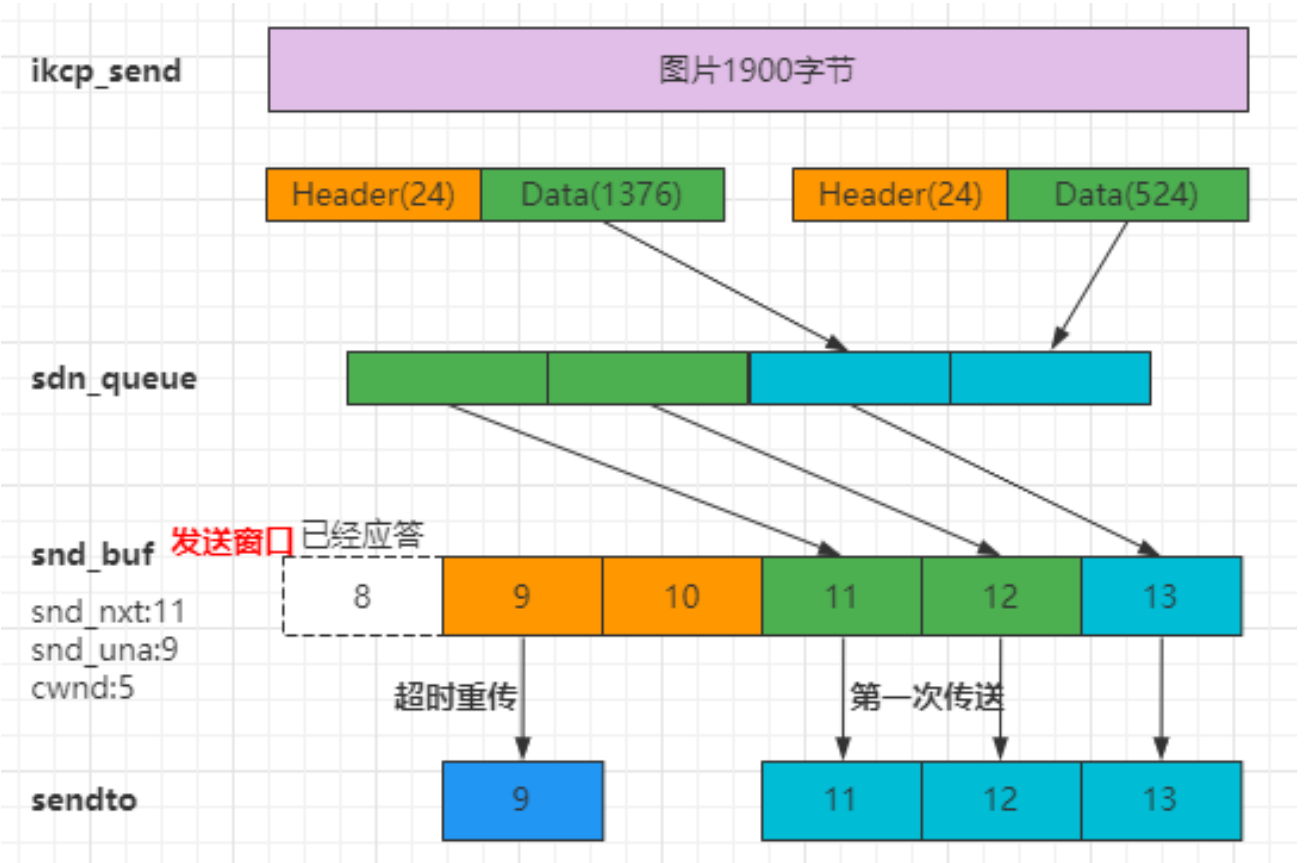
4.5 kcp协议头



cmd	作用
IKCP_CMD_PUSH	数据推送命令
IKCP_CMD_ACK	确认命令
IKCP_CMD_WASK	接收窗口大小询问命令
IKCP_CMD_WINS	接收窗口大小告知命令

IKCP_CMD_PUSH和IKCP_CMD_ACK 关联
IKCP_CMD_WASK和IKCP_CMD_WINS 关联

4.6 kcp发送数据过程



为什么需要send_queue
为什么需要snd_buf

4.6 kcp发送窗口

snd_wnd：固定大小，默认32
rmt_wnd：远端接收窗口大小，默认32
cwnd：滑动窗口，可变，越小一次能发送的数据越小

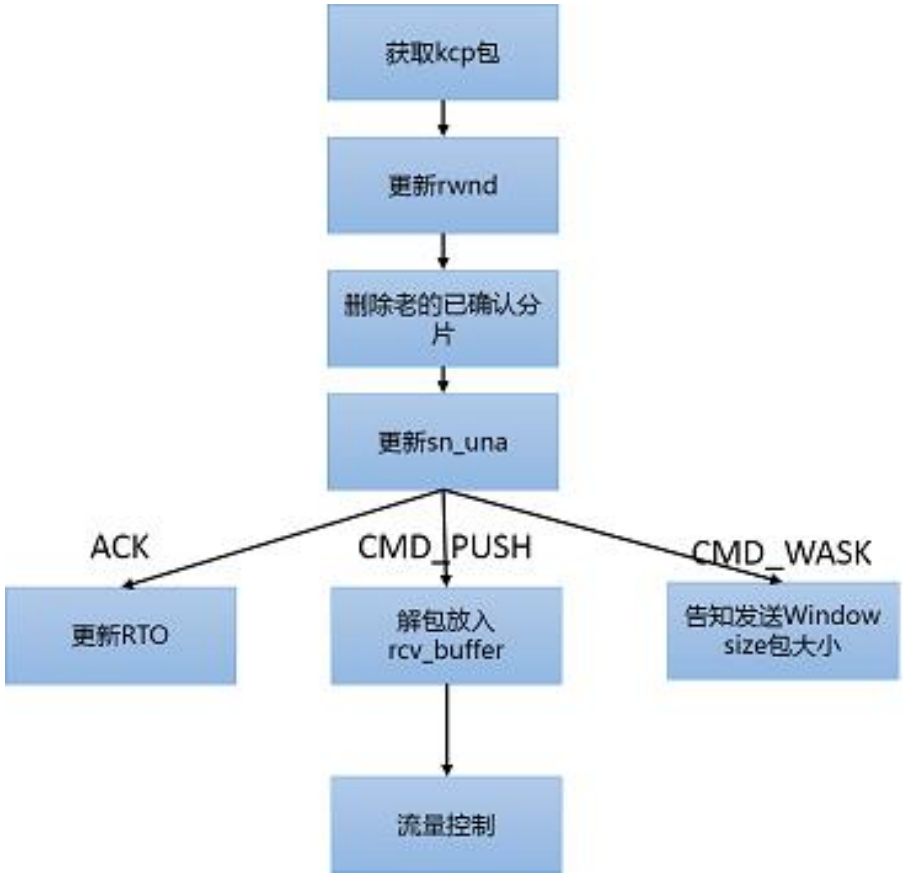
发送速率的控制是：本质是根据滑动窗口控制把数据从snd_ _queue 加入到send_buf 。

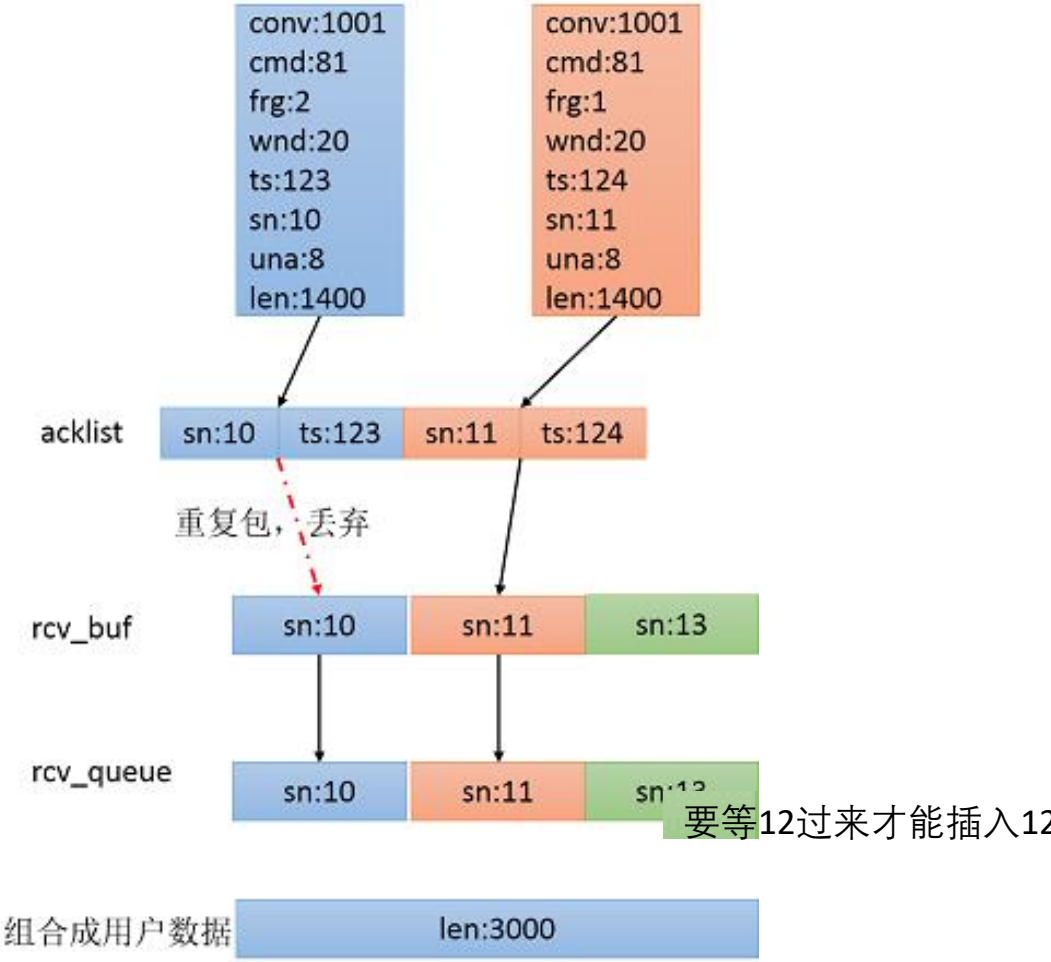
1. 调用void ikcp_flush(ikcpcb *kcp) 发送数据

IKCP_CMD_PUSH 命令

```
1116    // calculate window size 取发送窗口和远端窗口最小值得到拥塞窗口小
1117    cwnd = _imin_(kcp->snd_wnd, kcp->rmt_wnd);      // 当rmt_wnd为0的时候,
1118    // 如果做了流控制则取配置拥塞窗口、发送窗口和远端窗口三者最小值
1119    if (kcp->nocwnd == 0) cwnd = _imin_(kcp->cwnd, cwnd);   // 进一步控制cwnd大小
1120
1121    // move data from snd_queue to snd_buf
1122    // 从snd_queue移动到snd_buf的数量不能超出对方的接收能力   此时如果
1123    //  发送那些符合拥塞范围的数据分片
1124    while (_itimediff(kcp->snd_nxt, kcp->snd_una + cwnd) < 0) {
1125        IKCPSEG *newseg;
1126        if (iqueue_is_empty(&kcp->snd_queue)) break;
1127
1128        newseg = iqueue_entry(kcp->snd_queue.next, IKCPSEG, node);
1129
1130        iqueue_del(&newseg->node);
1131        iqueue_add_tail(&newseg->node, &kcp->snd_buf);  // 从发送队列添加到发送缓存
1132        kcp->nsnd_que--;
1133        kcp->nsnd_buf++;                                加入到发送队列
1134        //设置数据分片的属性
1135        newseg->conv = kcp->conv;
1136        newseg->cmd = IKCP_CMD_PUSH;
1137        newseg->wnd = seg.wnd;
1138        newseg->ts = current;
1139        newseg->sn = kcp->snd_nxt++;        // 序号
1140        newseg->una = kcp->rcv_nxt;
1141        newseg->resendts = current;
1142        newseg->rto = kcp->rx_rto;
1143        newseg->fastack = 0;
1144        newseg->xmit = 0;
1145    }
1146
```

4.7 kcp接收数据过程





要等12过来才能插入12

为什么需要rcv_queue
为什么需要rcv_buf

4.7 接收窗口

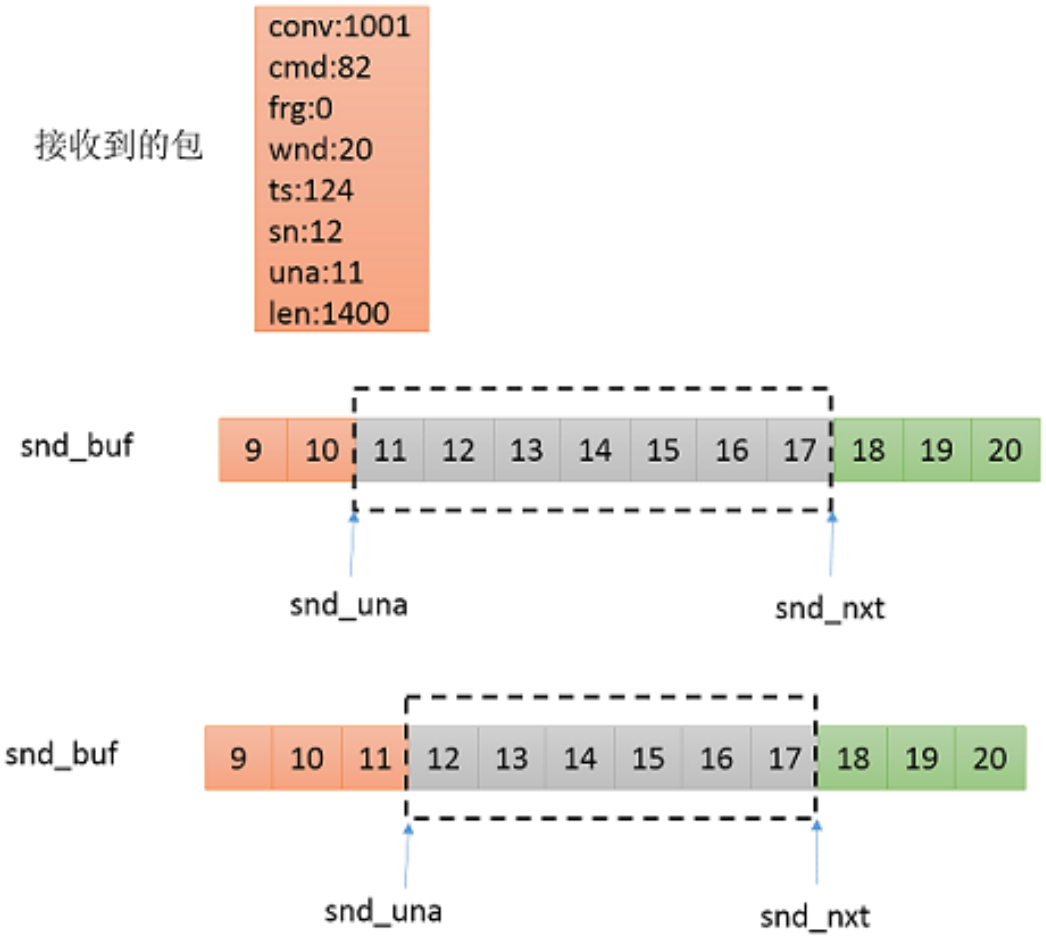
snd_wnd：固定大小，默认32
rmt_wnd：远端接收窗口大小，默认32
cwnd：滑动窗口，可变，越小一次能发送的数据越小

接收窗口的控制是：recv_queue的接收能力，比如默认接收端口为32，如果recv_queue接收了32个包后则接收窗口为0，然后用户读走了32个包，则接收窗口变为32。

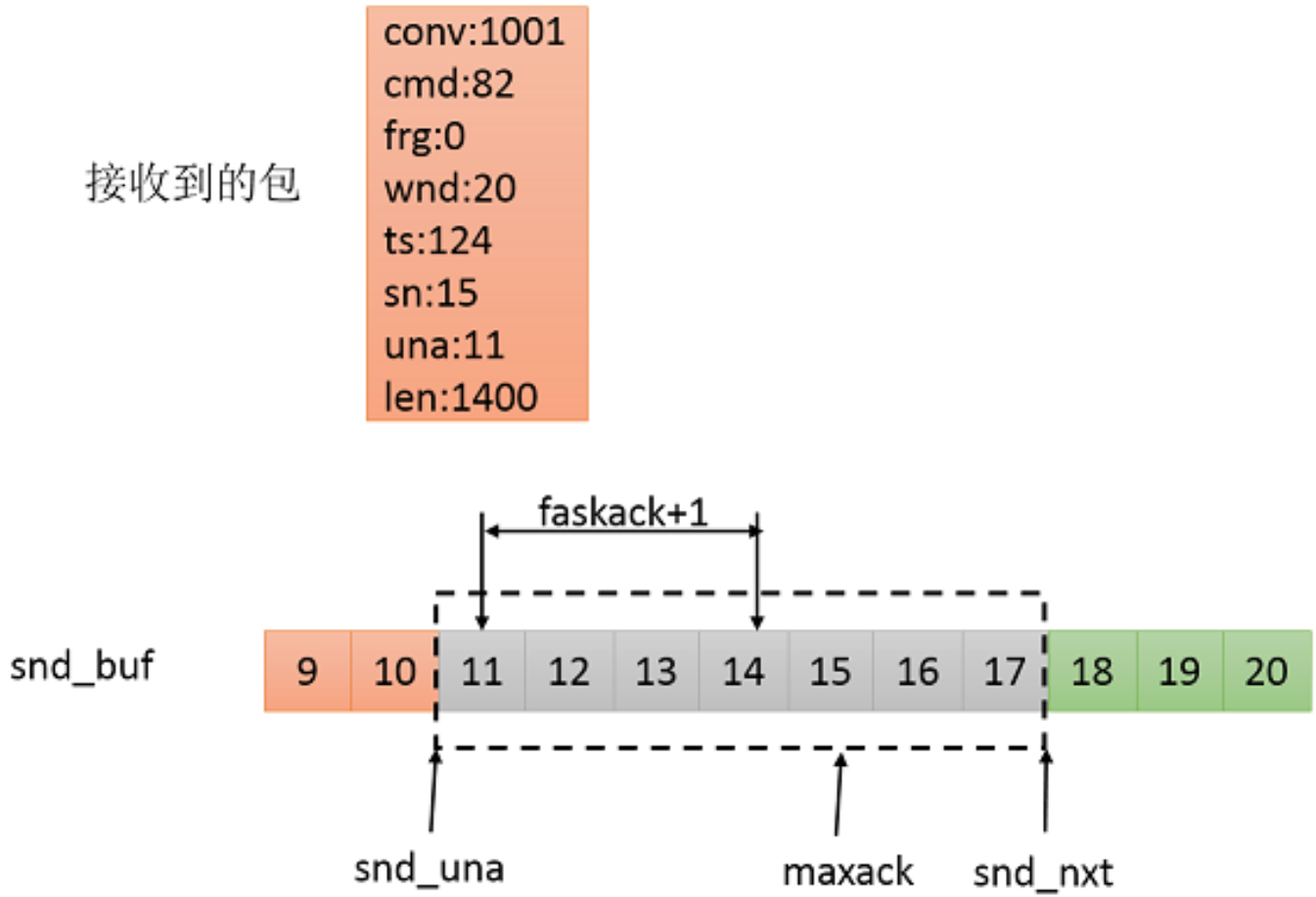
IKCP_CMD_PUSH 命令

```
996   // 计算可接收长度，以分片为单位
997   static int ikcp_wnd_unused(const ikcpcb *kcp)
998   {
999       if (kcp->nrcv_que < kcp->rcv_wnd) {
1000          return kcp->rcv_wnd - kcp->nrcv_que;
1001      }
1002      return 0;
1003  }
```

4.8 kcp确认包处理流程



4.9 kcp快速确认


接收到的包
conv:1001
cmd:82
frg:0
wnd:20
ts:124
sn:15
una:11
len:1400
faskack+1
snd_buf
9
10
11
12
13
14
15
16
17
18
19
20
snd_una
maxack
snd_nxt

4.10 应答列表acklist

- 收到包后将序号，时间戳存储到应答列表。

在ikcp_input函数调用ikcp_ack_push存储应答包

```
ikcp_ack_push(kcp, sn, ts); // 对该报文的确认 ACK 报文放入 ACK 列表中
```

- 发送应答包

在ikcp_flush函数发送应答包

- 应答包解析

在ikcp_input函数进行解析，判断IKCP_CMD_ACK

```
        kcp->rmt_wnd = wnd;                 // 携带了远端的接收窗口
        ikcp_parse_una(kcp, una);           // 删除小于snd_buf中小于una的segment，意思是una之前的都已经收到了
        ikcp_shrink_buf(kcp);     // 更新snd_una为snd_buf中seg->sn或kcp->snd_nxt ，更新下一个待应答的序号

        if (cmd == IKCP_CMD_ACK) {
            if (_itimediff(kcp->current, ts) >= 0) {        // 根据应答判断rtt
                //更新rx_srtt，rx_rttval，计算kcp->rx_rto
                ikcp_update_ack(kcp, _itimediff(kcp->current, ts));
            }
            //遍历snd_buf中（snd_una, snd_nxt），将sn相等的删除，直到大于sn
            ikcp_parse_ack(kcp, sn);        // 将已经ack的分片删除
            ikcp_shrink_buf(kcp);           // 更新控制块的 snd_una
            if (flag == 0) {
                flag = 1;        //快速重传标记
                maxack = sn;     // 记录最大的 ACK 编号
                latest_ts = ts;
            }   else {
                if (_itimediff(sn, maxack) > 0) {
                #ifndef IKCP_FASTACK_CONSERVE
```

这里是对方排序好的最大序号

这里是针对每个包的应答

4. 11 流量控制和拥塞控制

RTO计算（与TCP完全一样）

RTT：一个报文段发送出去，到收到对应确认包的时间差。

SRTT(kcp->rx_srtt)：RTT的一个加权RTT平均值，平滑值。

RTTVAR(kcp->rx_rttval)：RTT的平均偏差，用来衡量RTT的抖动。

4.12 探测对方接收窗口

ikcp_flush 发送探测窗口IKCP_CMD_WASK

```
1092    // flush window probing commands
1093    if (kcp->probe & IKCP_ASK_SEND) {
1094        seg.cmd = IKCP_CMD_WASK;        // 窗口探测  [询问对方窗口size]
1095        size = (int)(ptr - buffer);
1096        if (size + (int)IKCP_OVERHEAD > (int)kcp->mtu) {
1097            ikcp_output(kcp, buffer, size);
1098            ptr = buffer;
1099        }
1100        ptr = ikcp_encode_seg(ptr, &seg);
1101    }
```

探测窗口

ikcp_input函数

cmd == IKCP_CMD_WASK，标记kcp->probe |= IKCP_ASK_TELL;

ikcp_flush 回应探测IKCP_CMD_WINS

```
// flush window probing commands
if (kcp->probe & IKCP_ASK_TELL) {
    seg.cmd = IKCP_CMD_WINS;    // [告诉对方我方窗口size]，如果不为0，可以往我方发送数据
    size = (int)(ptr - buffer);
    if (size + (int)IKCP_OVERHEAD > (int)kcp->mtu) {
        ikcp_output(kcp, buffer, size);
        ptr = buffer;
    }
    ptr = ikcp_encode_seg(ptr, &seg);
}
```

4. 13 如何在项目中集成kcp

见课上代码演示

1. 范例演示
2. 如何集成到项目种，参考asio_kcp

```
#define ASIO_KCP_CONNECT_PACKET "asio_kcp_connect_package get_conv"
#define ASIO_KCP_SEND_BACK_CONV_PACKET "asio_kcp_connect_back_package get_conv:"
#define ASIO_KCP_DISCONNECT_PACKET "asio_kcp_disconnect_package"
```

5.0 QUIC时代是否已经到来

参考《 2-QUIC协议.pdf》

参考文档

通俗易懂讲解TCP流量控制机制，了解一下

5分钟读懂拥塞控制