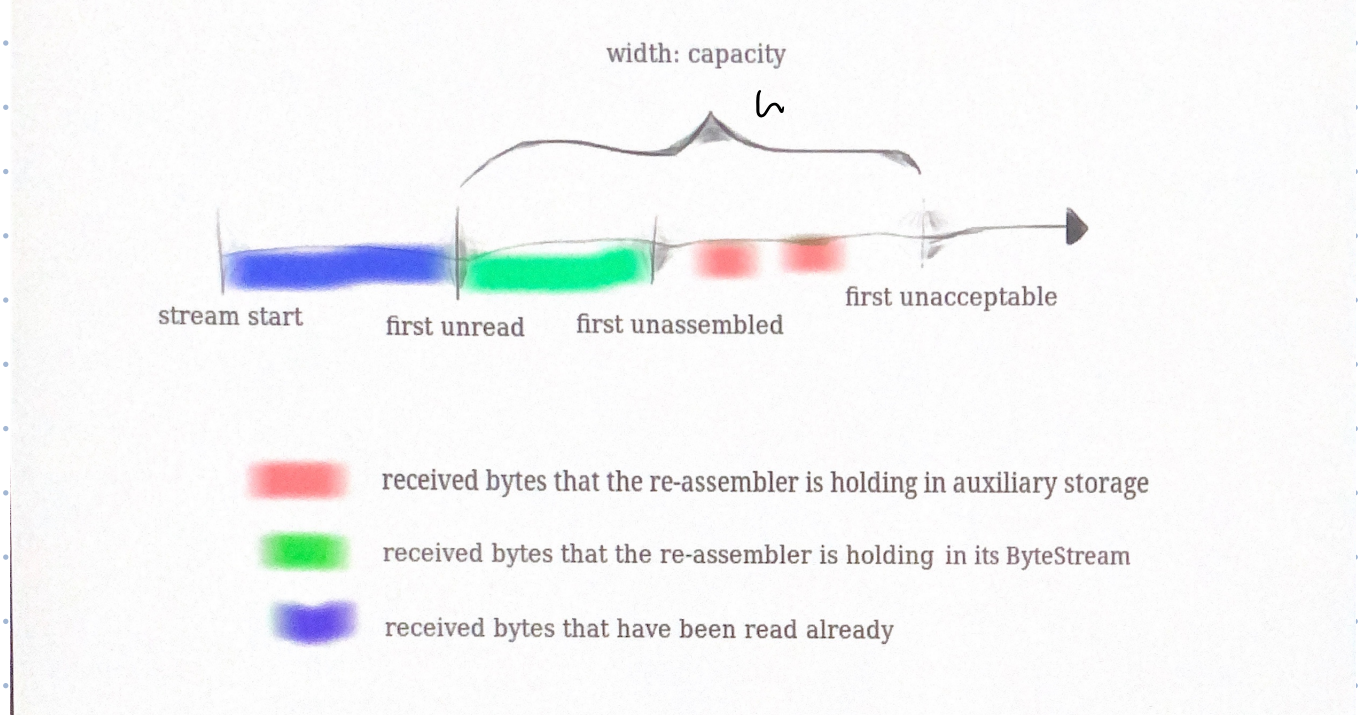
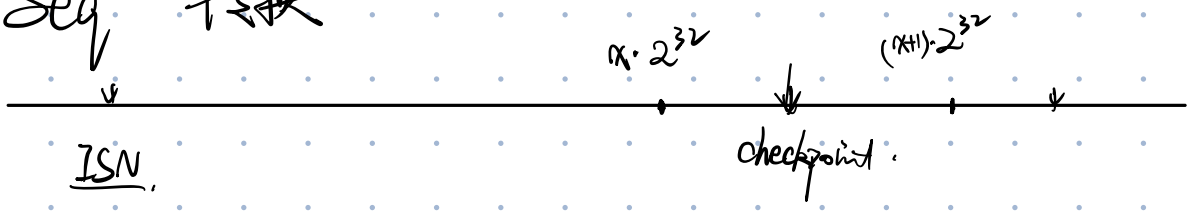
LAB 0.



搞懂这张图，迎刃而解。

LAB 1.

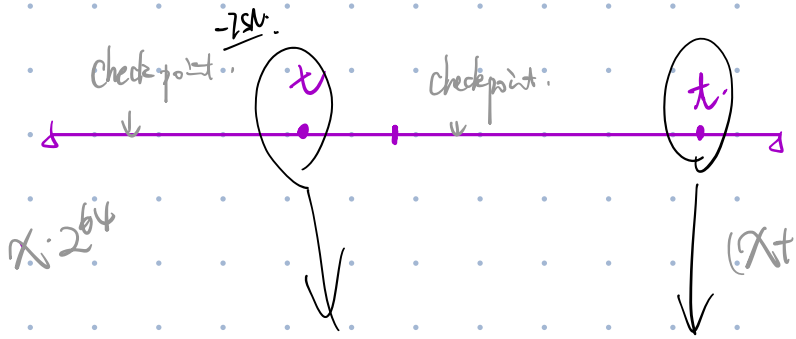
Seq 转换



$$n + \underbrace{n \cdot 2^{32}}_{t.} - ISN = \underline{ANS}.$$

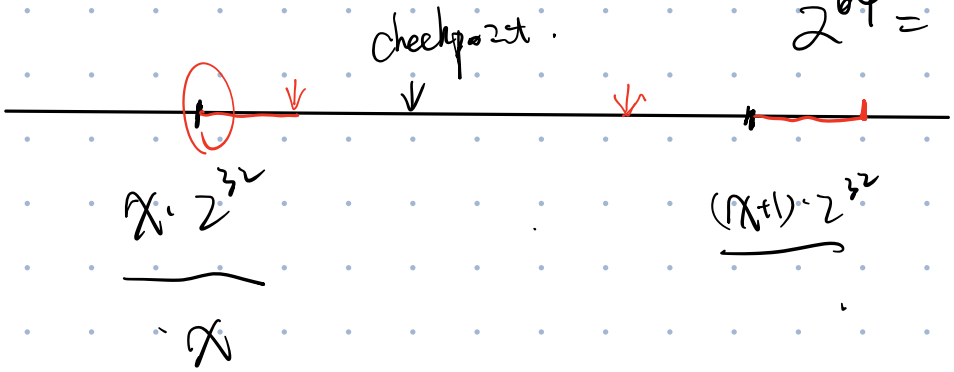
checkpoint. - ISN

ANS.



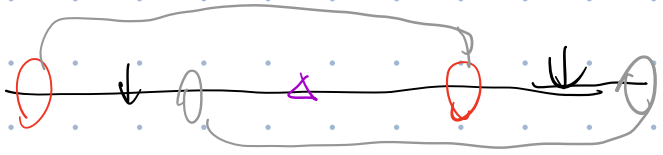
offset +

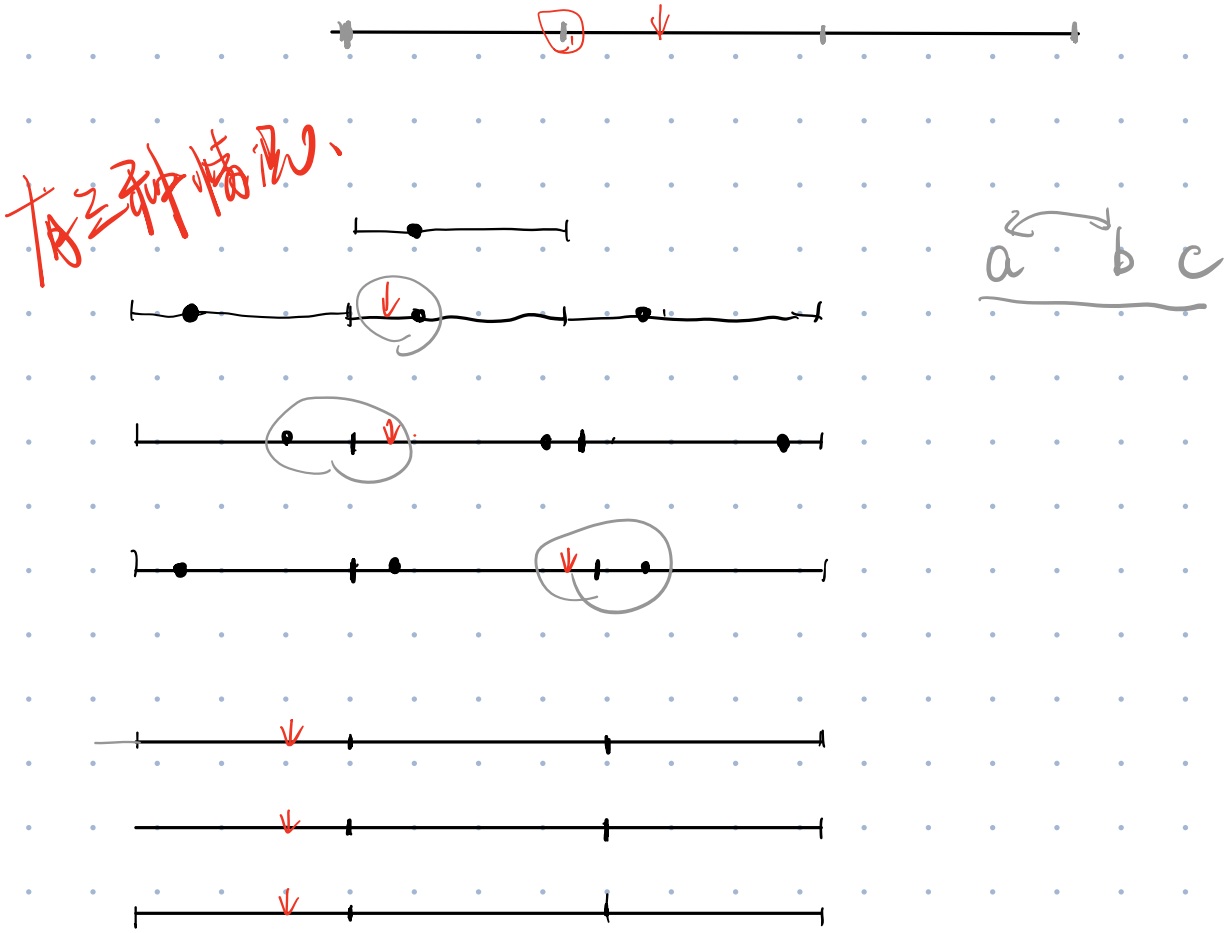
$2^{64} = 2^{32} \cdot 2^{32}$



$(ISN + Y) = n \cdot + n \cdot 2^{32}$.

$Y = offset + \underbrace{n}_{(n+1)} 2^{32}$.





element	SYN	c	a	t	FIN
seqno	$2^{32}-2$	$2^{32}-1$	0	1	2
absolute seqno	0	1	2	3	4
stream index		0	1	2	

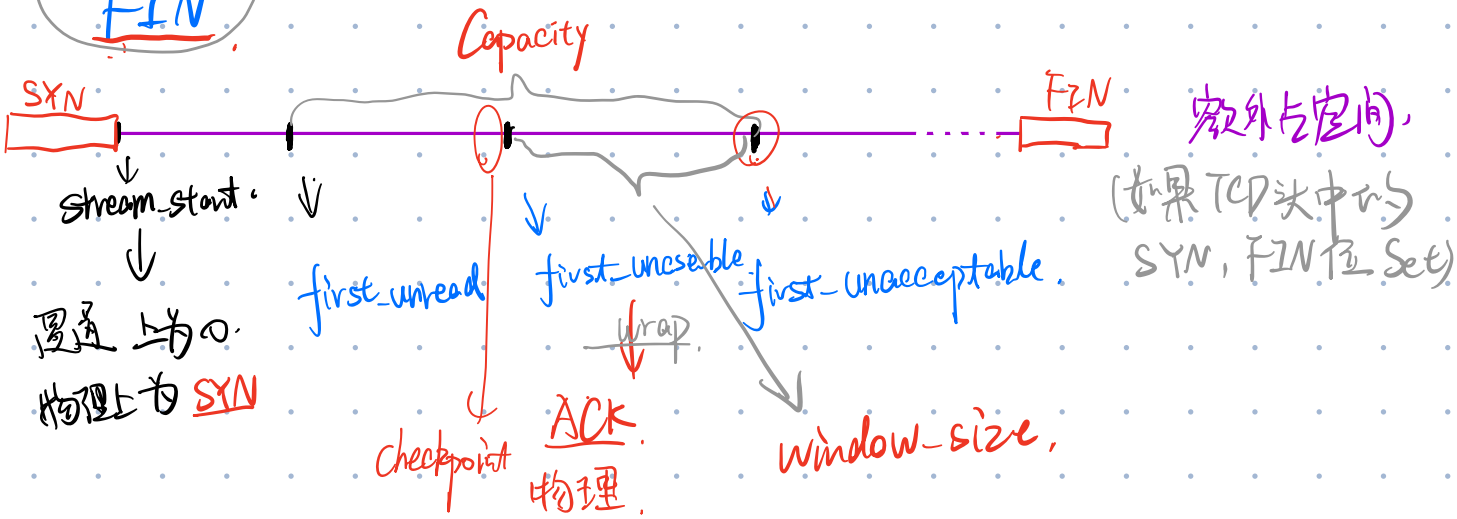
The figure shows the three different types of indexing involved in TCP:

Sequence Numbers	**Absolute Sequence Numbers**	**Stream Indices**
• Start at the ISN	• Start at 0	• Start at 0
• Include SYN/FIN	• Include SYN/FIN	• Omit SYN/FIN
• 32 bits, wrapping	• 64 bits, non-wrapping	• 64 bits, non-wrapping
• "seqno"	• "absolute seqno"	• "stream index"

LAB 2. TCP Reciever.

- TCP 中的 Index $[0, 2^{32}-1)$. ↘ 会 Wrap 转换.
  Reasembler 中 Idx → $[0, 2^{64}-1)$ ↙

- ISN 32位: 起始的序列号.
  SYN
  FIN
  各自额外占用 1个 SeqNumber. (32bit).

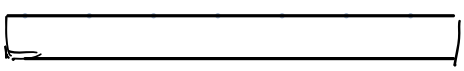


- Segment recieved 收到 xx个 片段.

FIN与 ACK 同时出现 → Ack 是多少

看是否传给重组器之后, 他 input_ended() 了.

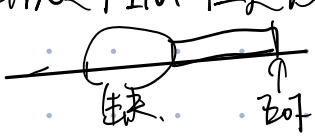
①

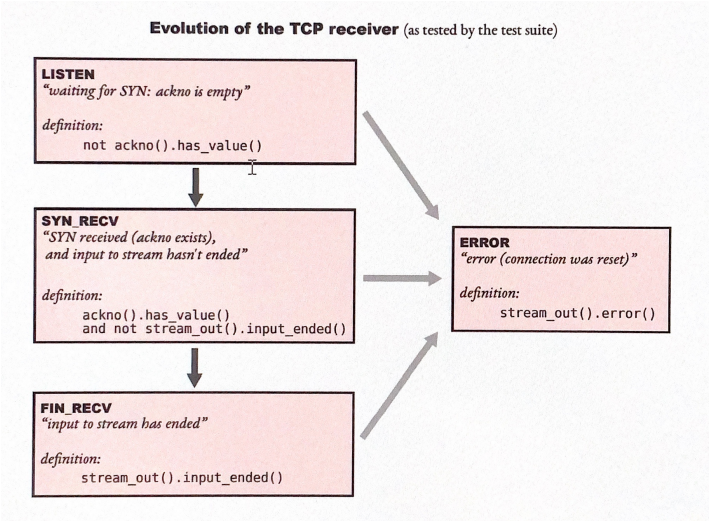


Ack = last_unsenble + SYN + FIN

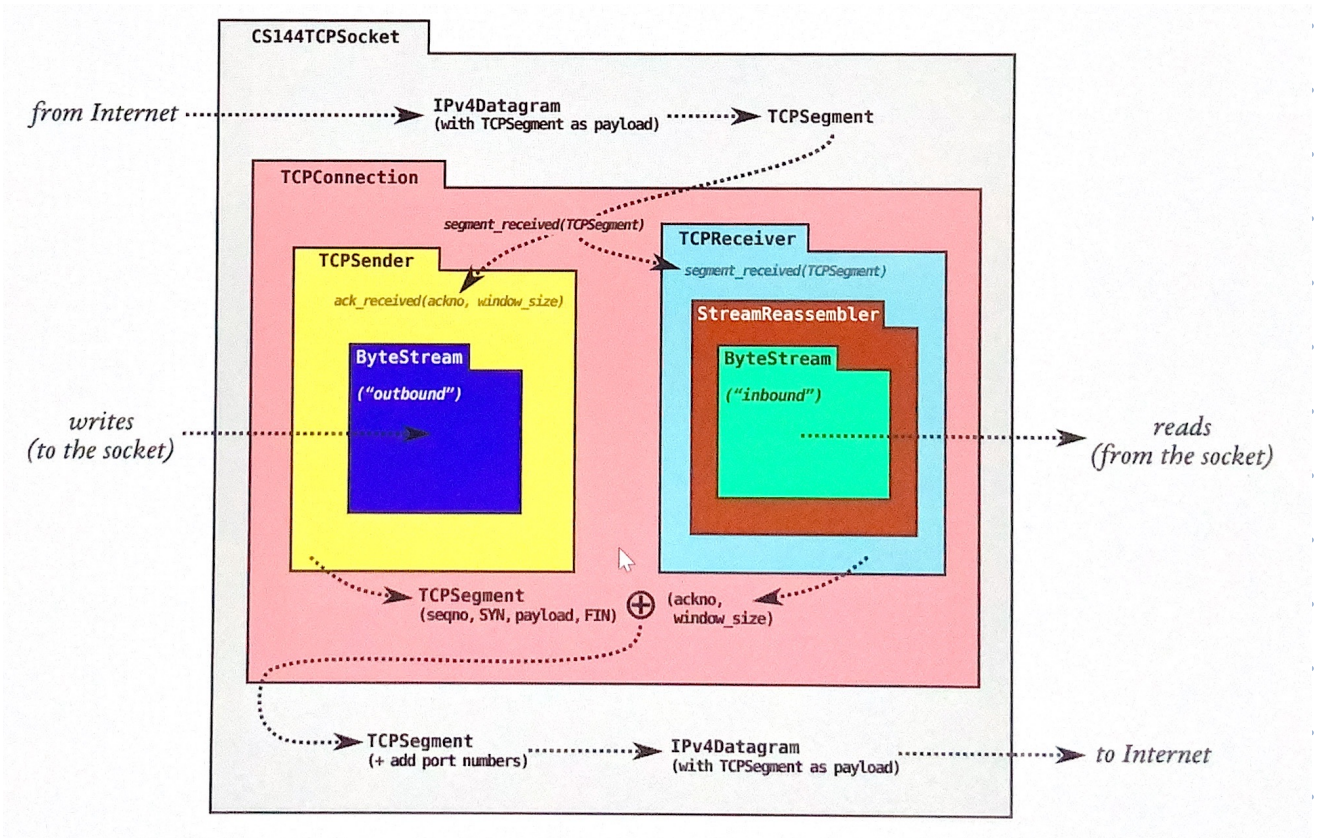
seq=0 完全读完了

可能之前出现过FIN, 但还没重组.





- 需要把乱序的传给 re_assembler.
- 但FIN还不是True，还没收FIN_Rev.
- Abs_idx ⇕ seq_idx 差1（FIN）



LAB 3. TCP Sender.

TCP Sender
- read
  - ackno
  - window_size.
- write:
  - seq.No
  - SYN - flag
  - FIN
  - payload

从 ByteStream 读. → 什么时候读/还要怎么读

写入 Segment. → 设置好 header 相关信息.

发送 → track. 是否有发了还没 ack 的.
超时. 没ACK, 重发.

△ ByteStrem 在传输时. 头尾会被 Sender 贴上 8-bit SYN / FIN flag.

△ tick() → 遍历所有 Outstanding Seg. → 最old是否还没 fully ACK.
收到的 ack < seq.

△ 超时.
- 自己实现 tick → tick(10) → 过了 10 ms
- RTO → 动态, 初始值不变. exp backoff

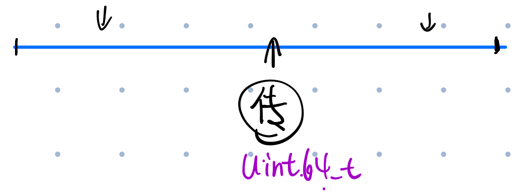
earliest. 有头, 还是有尾巴.

- 使用数据结构
  - 发了, 还没ACK的 Seg → 按照 seq 排序 (是seq?) 能用 ack. 访问?
  - 连续传送次数 → 比较上次和这次是否同一个Seg
    last_send_seg = ____
- 实现 timer.

start() 给出信号. 多线程?

Q: • 发的 Segment 里可能 数据长为 0. 此时 seqno 是多少?

Q unwrap → 收到的 ACK → abs_idx.



- timer 到时间了 → 重传
  ↳ 如果 window_size > 0.
  连续重传次数 +1

- 空包的作用.
  ↓
  获取新 win_size
  ackno

next_seqno_abs − last_recv_ackno > 0
= bytes_in_flight.

in the flight

fill_window
↓
bytes_in_flight = last_window.

- 重传不用管 window_size，之前已经计算过了

- C++ 形参 < 声明时 set 默认值 / 定义时 set —— ， 不能同时

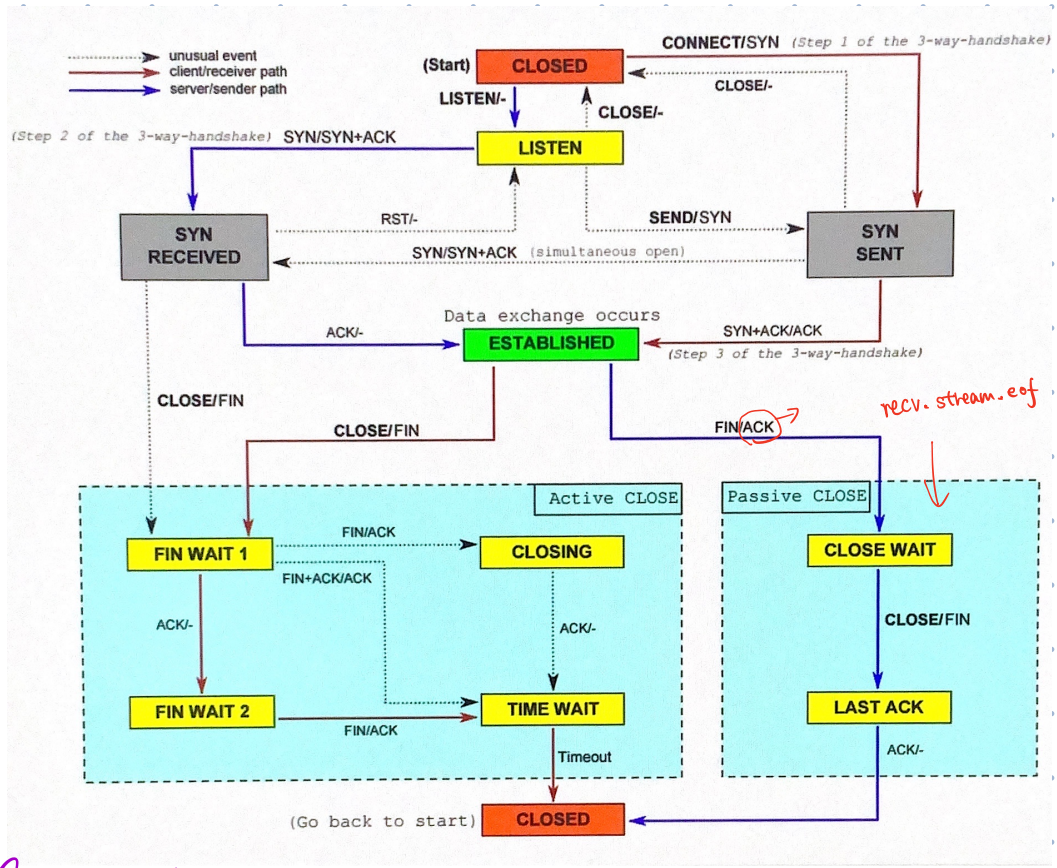
- FIN_FLAG 可以和 Data 一起传输

- ByteStream 有两种 < eof() / input_ended() + buffer_empty()

- ACK of new data → timer restarts,

- window_size == 0 → 置为 1.
  但不能将 RTO 翻倍.

Lab 4. TCP in Full.



TCP Connection.

让 Sender / Reciever 知道对应的 Seg 信息. (通知)

调用方法. 让 Sender / Redever 修改 Seg 信息.

Recieve.

RST ?

Recver: segment_recieved( )

ACK ? → Sender ← ackno / window.

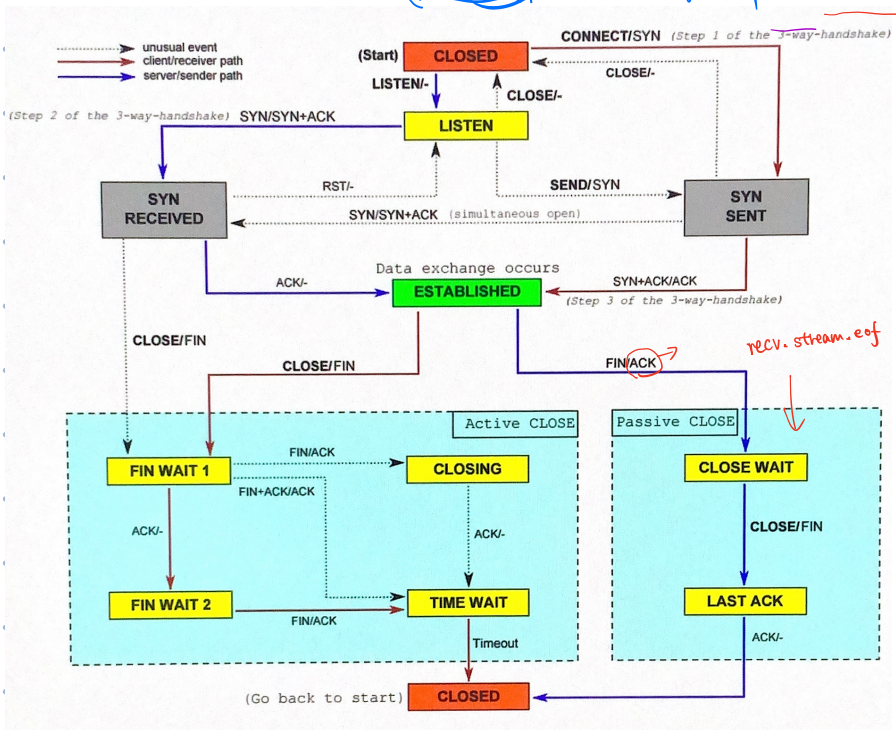
如果 in_seq 有 seqno, 至少发 1 个 segment. 回应.

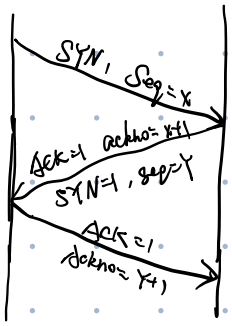
判断

Sending { Sender : Seqno / SYN / FIN / PAYLOAD
Reciever : Ackno / win_size.

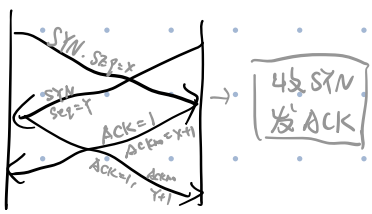
Connection
- Tick { 通知 Sender 时间流逝
重传 limit → > MAX_RETX_ ...
- 是否 Alive? { either stream is running.
还在 lingering.
- inbound
outbound

State → 判断此时处于何种状态 (Robust?)



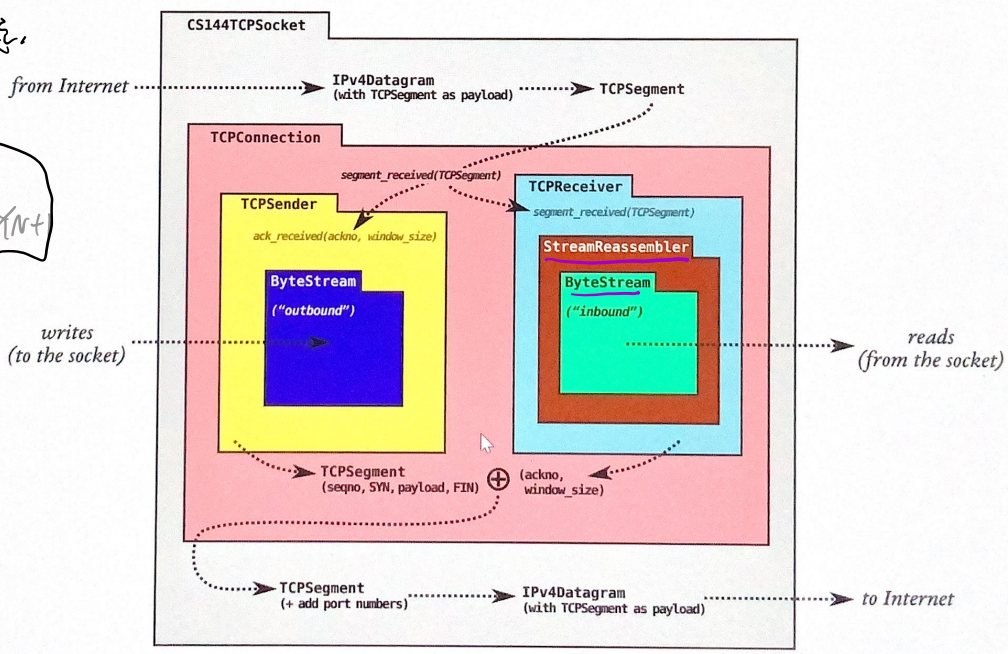


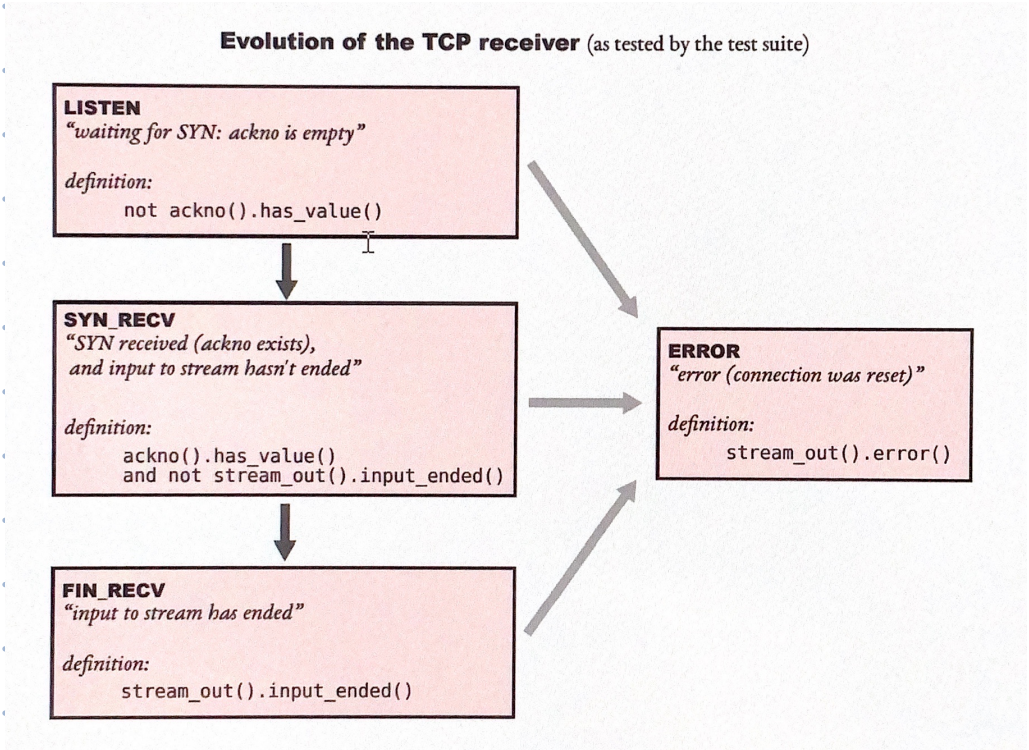
同时打开?

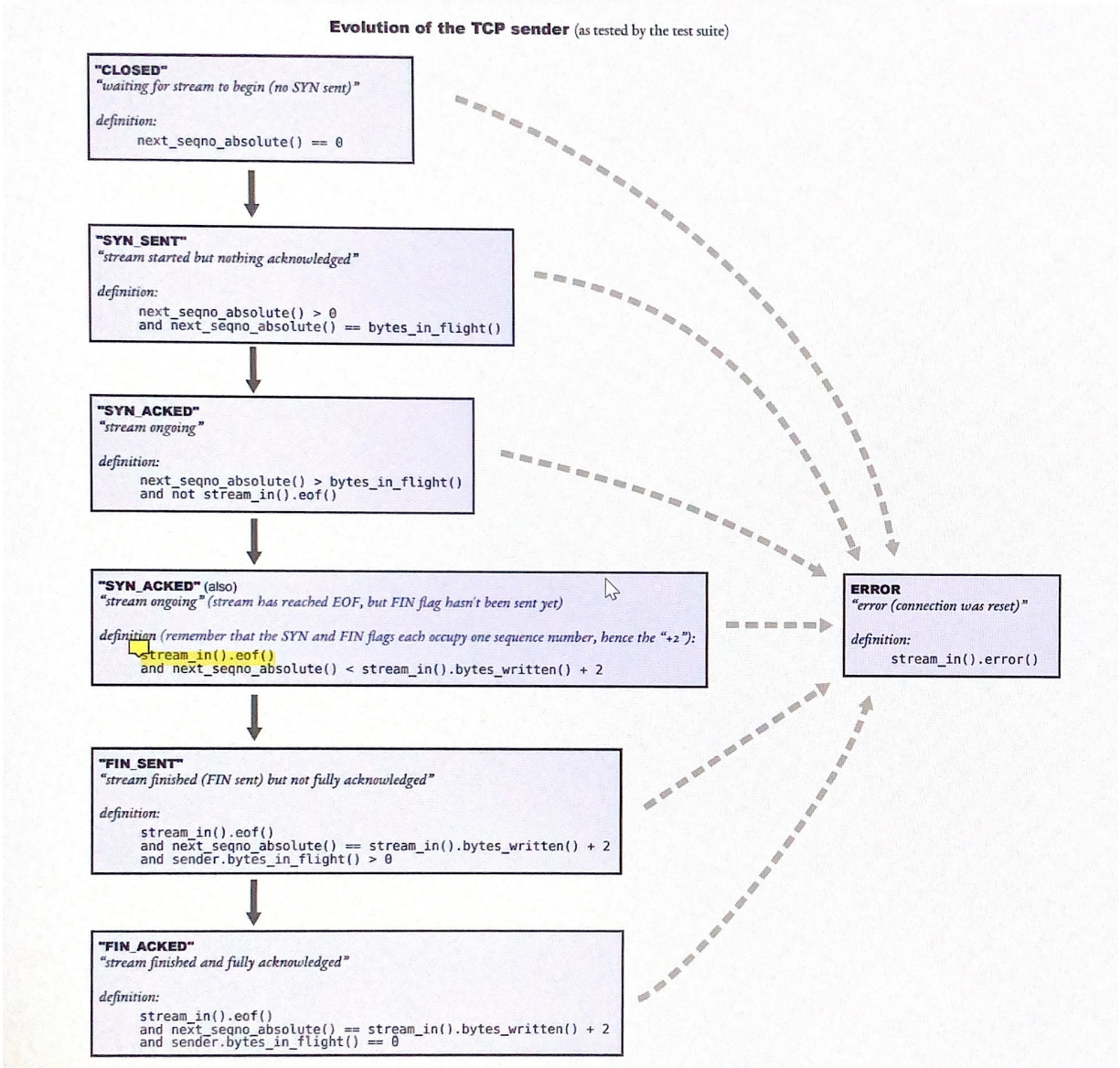


- Connect
主：发 SYN, seq=ISN
被：发 SYN, seq=ISN.
有 SYN → 回 ACK=1, ackno= SYN+1
ACK= ISN+1 → 建立.

可选.







Active.

发/发一个 RST Seg.

unclean cooldown

(error state.
+
active

"CLOSED"
"waiting for stream to begin (no SYN sent)"

definition:

```
next_seqno_absolute() == 0
```

"SYN_SENT"
"stream started but nothing acknowledged"

definition:

```
next_seqno_absolute() > 0
and next_seqno_absolute() == bytes_in_flight()
```

"SYN_ACKED"
"stream ongoing"

definition:

```
next_seqno_absolute() > bytes_in_flight()
and not stream_in().eof()
```

"SYN_ACKED" (also)
"stream ongoing" (stream has reached EOF, but FIN flag hasn't been sent yet)

definition (remember that the SYN and FIN flags each occupy one sequence number, hence the "+2"):

```
stream_in().eof()
and next_seqno_absolute() < stream_in().bytes_written() + 2
```

"FIN_SENT"
"stream finished (FIN sent) but not fully acknowledged"

definition:

```
stream_in().eof()
and next_seqno_absolute() == stream_in().bytes_written() + 2
and sender.bytes_in_flight() > 0
```

"FIN_ACKED"
"stream finished and fully acknowledged"

definition:

```
stream_in().eof()
and next_seqno_absolute() == stream_in().bytes_written() + 2
and sender.bytes_in_flight() == 0
```

LISTEN
"waiting for SYN: ackno is empty"

definition:

```
not ackno().has_value()
```

SYN_RECV
"SYN received (ackno exists), and input to stream hasn't ended"

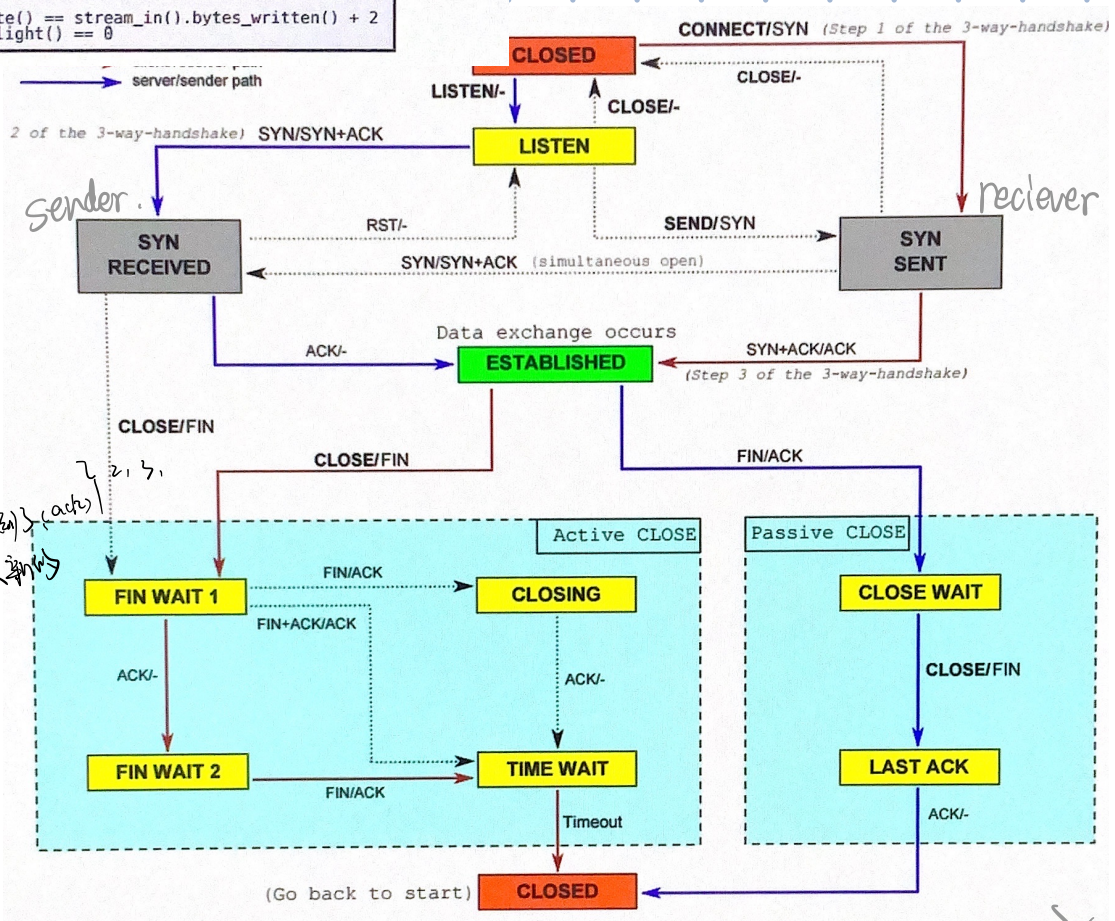
definition:

```
ackno().has_value()
and not stream_out().input_ended()
```

FIN_RECV
"input to stream has ended"

definition:

```
stream_out().input_ended()
```



TCP 要何时断开连接？ → clean cooldown. 返回 false.

① · Reiver 的 inbound 流 end 了， eof；
② · Sender 的 outbound stream 结束了， eof.
  ↓ FIN 发出了 → FIN
③ · Sender 发的 包括 FIN 都被 ack 了。 FIN_Ack. nxt_abs_seqno = writen+2

都满足了 已经（自己）· Condition 对方(peer) 已经满足

A. in/out 流都 end 之后， lingering
 10x initial_RTO 内 没收到 任何 Seg / from peer
 主动关闭

B. 自己满足 1-3，
 100% 确定 peer 满足了；→ remote 先 end_stream

收到 Remote 的 FIN，要确认对方的 FIN，要发一个 自己 历史最大的 seqno / 加上 Ackno. （至少是FIN） Seg 去确认收到的FIN
先收Remote的FIN.
remote 确认这个 Seg.
则自己的 ③ 可以保证。

自己的 ③ 满足 + 先收到 FIN.
 ↓ ACK 比 FIN 要大
自己发的确认FIN的Seg，被对方 Ack
 ↓
对方知道它的 FIN 被自己确认了
 ↓
对方的 ③ 满足。

所有发的包被确认。

最直接的： inbound_stream（Receiver）在自己（Sender）未发送 FIN 时，就结束了。
 → 这样不用再 linger.
 ↓
inbound stream 在 outbound_stream 达到 EOF 之前，先 end 了。

- 收到RST → ① 设 in/out_bound 为 error_flag.
  ② active_false.

发送RST → when?

① 重传次数过多.

② 析构函数在 active() 时被调用.

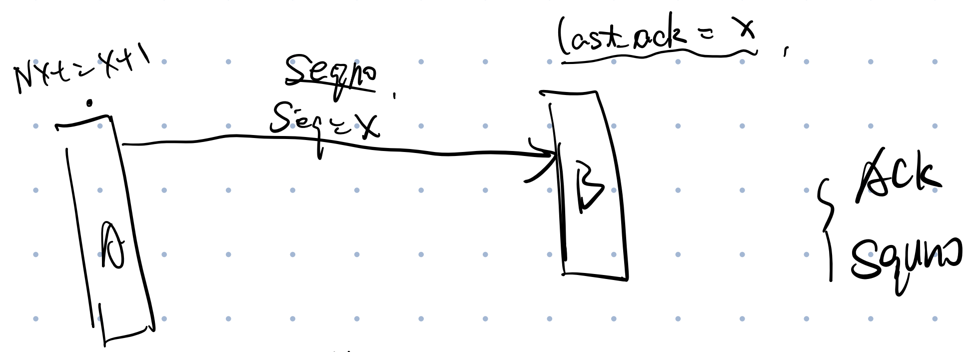
effect → 同收到 ①/②.

↓

空seg + RST_flag.

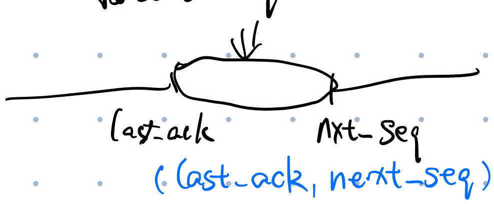
- 如果 window_size > uint16 → 忽略, 发最大的值.

empty_seg → {
- seqno - invalid
- 无ACK
- → window.



发 { ackno - 确认之前的...
win_size - 是否有更新 (win_size)

接受的 seq



收 { seq → 序号 → 用来重组.
ACK → 当前已重组到 ACK-1.

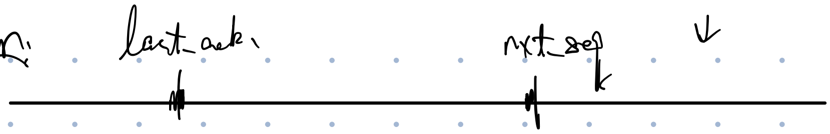
↓ 写入 seq 发出.

遇到 empty，你 set 一个 (last_ack, next_seq) 的值。 (ACK)

↳ seq。(remote peer 的 last_ack - 1 。

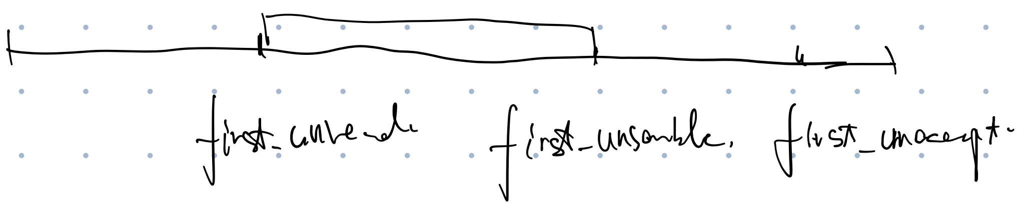
刚开始，_reciever 没有 ackno 可以提供

sender:



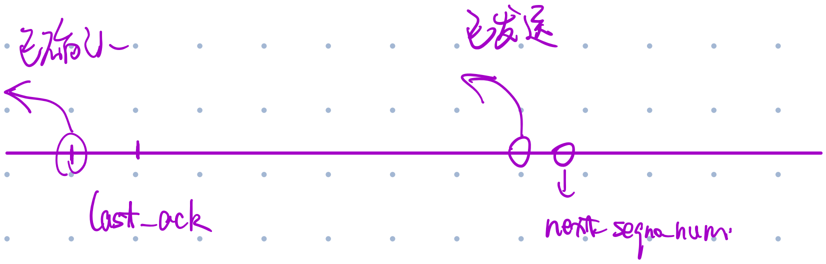
- 没 ack：
  - ackno → nxtseq 还没发出
  - seqno → 在可接收范围外

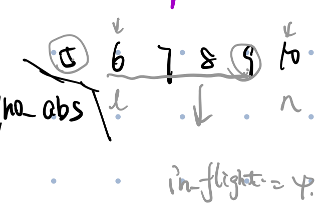
reciever:



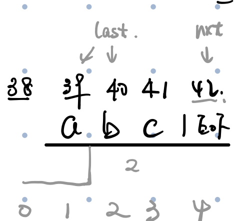
close_file 里的 fin → 结束   × 一样的

和 sender 里的 fin 被另一个 → 另外的 fin





_memo_size ? next_seqno_abs



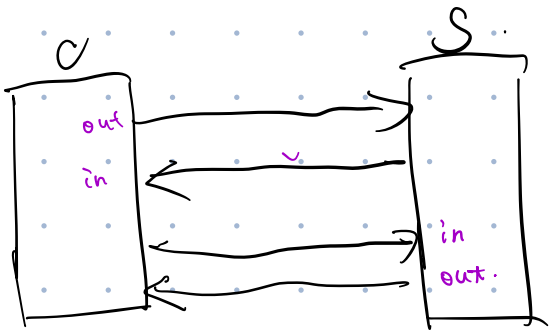
问题，memo. 不是逐 byte buffer 的
一次只能移除一整块。
只有整个块完全 ack 了才能 pop.

① ② ③ ④

70 71 72 73 74 75 76
s a b c d e f
0 1 2 3 4 5 6

Zero_window的处理. win_size → 清出

用window为负的处理.



Fin wait 1/2 的时候, sender outbound stream关闭,
但还需要发送ACK, 直至对方的FIN发来

keep alive包 — win=0

{ 纯Ack包 — seq = ack-1, len=0

| 0-window probe包 — seq=ack, len=1

→ 纯ack包 Win=0

→