网络编程第一次作业

吴桐
57119108

2022 年 9 月 14 日

1 基于虚拟机的 TCP 建立及终止过程

根据示范代码，我们尝试在一台虚拟机上同时启动服务器和客户端程序，建立 TCP 连接。我们先启动服务器程序，之后启动客户端程序，发现出现以下两种错误：segmentation error 和 connect error。接下来我们将解释并解决这两个错误。

我们发现在服务器端显示段错误（segmentation error），在逐句排查后发现程序是在获取时间的语句处退出了。我们修改代码如下：

```
time( &ticks );
struct tm *p;
p = gmtime( &ticks );
```

由于虚拟机的特殊性，我们不能使用固定端口，因此需要把程序修改成用户自定义端口号。在实验时我们优先选择 4000 之后的端口。

最后一个问题就是，TCP 连接的建立会“概率性”地失败。如图 1 所示，我们一开始可以正常建立 TCP 连接，查看此时的端口进程信息，发现服务器进程正常监听端口。当我们关闭服务器后，再次重启服务器，尝试从客户端建立连接，此时程序提示“connect error”。查看端口进程信息，发现该端口处于 $TIME\_WAIT$ 状态。在等待一段时间后，端口结束了 $TIME\_WAIT$ 状态，此时我们重启服务器，可以发现服务器进程正常监听端口。

$TIME\_WAIT$ 状态为客户端在结束连接后进入的正常状态，需要等待 2MSL 后才能返回到 CLOSED 状态。在此状态下端口会被占用，因此建立 TCP 连接失败。这正是在一台主机上同时启动服务器和客户端会引发的特殊问题。

我们在程序中添加端口复用代码，就可以解决此问题。

```
int reuse = 1;
if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(int)) < 0)
{
```

```
    printf("setsockopet error\n");
    exit( 1 );
}
```

如图 2 所示，我们一开始可以正常建立 TCP 连接，查看此时的端口进程信息，发现服务器进程正常监听端口。当我们关闭服务器后，可以看到该端口处于 $TIME\_WAIT$ 状态。立即重启服务器程序，可以看到服务器进程正常监听端口，此时能够正常建立 TCP 连接。

```
[09/07/22]seed@VM:~/.../Network Programming$ a.out 0.0.0.0 4433
Wed Sep  7 23:02:55 2022
[09/07/22]seed@VM:~/.../Network Programming$ sudo netstat -anp | grep 4433
tcp        0      0 0.0.0.0:4433            0.0.0.0:*               LISTEN      5077/b.out

[09/07/22]seed@VM:~/.../Network Programming$ a.out 0.0.0.0 4433
Wed Sep  7 23:06:28 2022
[09/07/22]seed@VM:~/.../Network Programming$ a.out 0.0.0.0 4433
connect error
[09/07/22]seed@VM:~/.../Network Programming$ sudo netstat -anp | grep 4433
tcp        0      0 127.0.0.1:4433          127.0.0.1:47872         TIME_WAIT   -

[09/07/22]seed@VM:~/.../Network Programming$ a.out 127.0.0.1 4433
connect error
[09/07/22]seed@VM:~/.../Network Programming$ sudo netstat -anp | grep 4433
tcp        0      0 127.0.0.1:4433          127.0.0.1:47872         TIME_WAIT   -

[09/07/22]seed@VM:~/.../Network Programming$ sudo netstat -anp | grep 4433
[09/07/22]seed@VM:~/.../Network Programming$ sudo netstat -anp | grep 4433
tcp        0      0 0.0.0.0:4433            0.0.0.0:*               LISTEN      5104/b.out
```

图 1: 无端口复用功能的服务器代码运行结果

```
[09/07/22]seed@VM:~/.../Network Programming$ a.out 0.0.0.0 4433
Wed Sep  7 23:18:31 2022
[09/07/22]seed@VM:~/.../Network Programming$ sudo netstat -anp | grep 4433
tcp        0      0 0.0.0.0:4433            0.0.0.0:*               LISTEN      5421/b.out

tcp        0      0 127.0.0.1:4433          127.0.0.1:47884         TIME_WAIT   -

[09/07/22]seed@VM:~/.../Network Programming$ sudo netstat -anp | grep 4433
tcp        0      0 127.0.0.1:4433          127.0.0.1:47884         TIME_WAIT   -

[09/07/22]seed@VM:~/.../Network Programming$ sudo netstat -anp | grep 4433
tcp        0      0 0.0.0.0:4433            0.0.0.0:*               LISTEN      5436/b.out

tcp        0      0 127.0.0.1:4433          127.0.0.1:47884         TIME_WAIT   -

[09/07/22]seed@VM:~/.../Network Programming$ a.out 0.0.0.0 4433
Wed Sep  7 23:18:56 2022
```

图 2: 添加端口复用功能的服务器代码运行结果

2 调整 TCP 缓冲区

TCP 连接是由内核维护的，内核为每个连接建立的内存缓冲区，既要为网络传输服务，又要充当进程与网络间的缓冲桥梁。如果连接的内存配置过小，就无法充分使用网络带宽，TCP 传输速度就会很慢；如果连接的内存配置过大，那么服务器内存会很快用尽，新连接就无法建立成功。

我们知道 TCP 提供了可靠的传输，主要的机制就是在报文发出后，必须收到接收方返回的 ACK 确认报文，如果在 RTO 内还没收到，就会重新发送这个报文。由此可知，TCP 报文发出去后，并不能立刻从内存中删除，因为重发时还需要用到它。

网络传输接收方的处理能力有限，把它的处理能力告诉发送方，使其限制发送速度，这即是滑动窗口。TCP 头部设计了设计了 2 个字节的窗口字段，来起到通知发送方的作用。该字段最多只能表达 2^{16} 即 65535 字节大小的窗口，这在 RTT 为 10ms 的网络中也只能到达 6MB/s 的最大速度，在当今的高速网络中显然并不够用。RFC1323 中定义了扩充窗口的方法，在 Linux 中打开这一功能，需要把 $tcp\_window\_scaling$ 配置设为 1，此时窗口的最大值可以达到 1GB。

```
net.ipv4.tcp_window_scaling = 1
```

当然，尽管接收缓冲区配置的足够大，接收窗口无限放大，发送方也不可能无限的提升发送速度，因为网络的传输能力是有限的，当发送方依据发送窗口，发送超过网络处理能力的报文时，路由器会直接丢弃这些报文。因此，缓冲区的内存并不是越大越好。

在 socket 网络编程中，通过设置 socket 的 SO_SNDBUF 属性，就可以设定缓冲区的大小。

Linux 提供了的缓冲区动态调节功能，可以设置缓冲区的调节范围。对于发送缓冲区，它的范围通过 tcp_wmem 配置。

```
net.ipv4.tcp_wmem = 4096 16384 4194304
```

其中三个数值分别为动态范围的“下限，初始默认值，上限”，发送缓冲区自动调节的依据是待发送的数据，且发送缓冲区的调节功能是自动开启的。

而接收缓冲区可以依据空闲系统内存的数量来调节接收窗口。如果系统的空闲内存很多，就可以把缓冲区增大一些，这样传给对方的接收窗口也会变大，因而对方的发送速度就可以相应地提升。

发送缓冲区的调节功能是自动开启的,而接收缓冲区则需要配置 $tcp\_moderate\_rcvbuf$ 为 1 来开启调节功能：

```
net.ipv4.tcp_moderate_rcvbuf = 1
```

而接收缓冲区通过 tcp_mem 判断空闲内存的多少。

```
net.ipv4.tcp_mem = 88560 118080 177120
```

tcp_mem 的 3 个值，是 Linux 判断系统内存是否紧张的依据。当 TCP 内存小于第 1 个值时，不需要进行自动调节；在第 1 和第 2 个值之间时，内核开始调节接收缓冲区的大小；大于第 3 个值时，内核不再为 TCP 分配新内存，此时新连接是无法建立的。

在高并发服务器中，为了兼顾网速与大量的并发连接，我们应当保证缓冲区的动态调整上限达到带宽时延积，而下限保持默认的 4K 不变即可。而对于内存紧张的服务而言，可以调低默认值作为提高并发的手段。

3 TCP 建立过程中的数据包分析

No.	Time	Source	Destination	Protocol	Length	Info
1	2022-09-14 04:0…	127.0.0.1	127.0.0.1	TCP	76	39468 → 4433 [SYN] Seq=179110168
2	2022-09-14 04:0…	127.0.0.1	127.0.0.1	TCP	76	4433 → 39468 [SYN, ACK] Seq=5694
3	2022-09-14 04:0…	127.0.0.1	127.0.0.1	TCP	68	39468 → 4433 [ACK] Seq=179110168
4	2022-09-14 04:0…	127.0.0.1	127.0.0.1	TCP	94	4433 → 39468 [PSH, ACK] Seq=5694
5	2022-09-14 04:0…	127.0.0.1	127.0.0.1	TCP	68	39468 → 4433 [ACK] Seq=179110168
6	2022-09-14 04:0…	127.0.0.1	127.0.0.1	TCP	68	4433 → 39468 [FIN, ACK] Seq=5694
7	2022-09-14 04:0…	127.0.0.1	127.0.0.1	TCP	68	39468 → 4433 [FIN, ACK] Seq=1791
8	2022-09-14 04:0…	127.0.0.1	127.0.0.1	TCP	68	4433 → 39468 [ACK] Seq=569466372

图 3: TCP 建立及终止过程的数据包

如图 3 所示，为“TCP 连接建立-数据传输-TCP 连接结束”整个过程中涉及到的数据包。图 4 为数据传输过程中从服务器发送给客户端的数据包。此次通信的内容并没有进行加密，而是明文传送，因此我们可以看到服务器发给客户端的日期和时间的明文信息。

每个数据包的作用如下所示：

1. 客户端发送 SYN 报文，请求进行连接。
2. 服务器回复 ACK 报文。
3. 客户端回复 ACK 报文，TCP 连接建立的三次握手完成。
4. 服务器发送 PSH+ACK 报文，说明有真正的 TCP 数据包内容传输，即服务器上的日期和时间信息。
5. 客户端回复 ACK 报文。
6. 服务器发送 FIN 报文，主动关闭 TCP 连接。
7. 客户端回复 ACK 报文。
8. 服务器回复 ACK 报文，TCP 连接关闭。

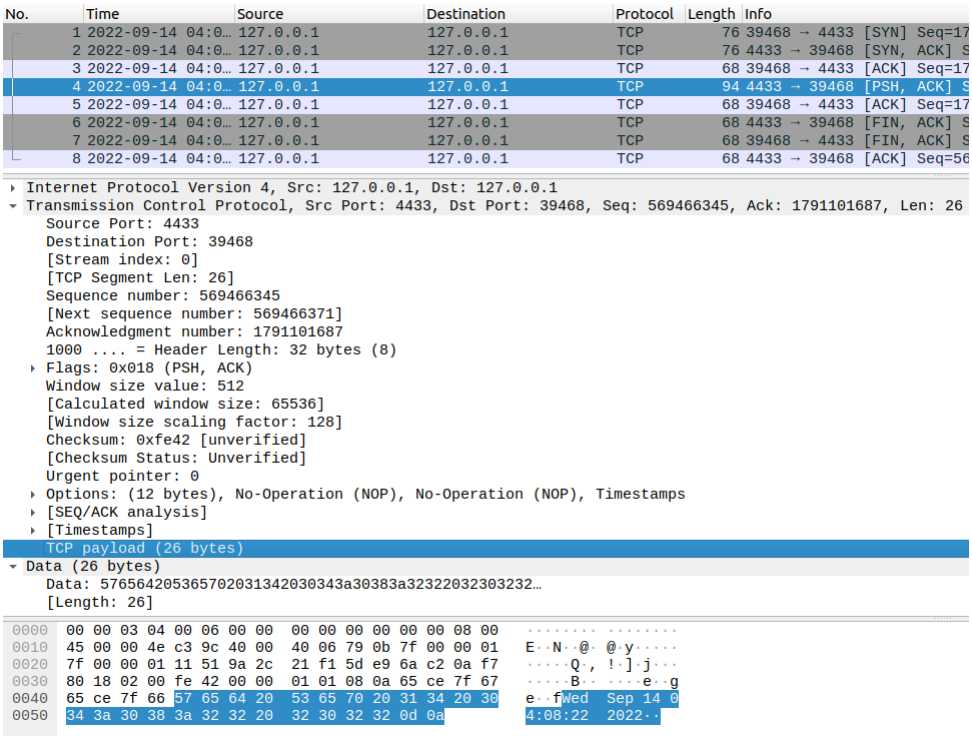
No. Time Source Destination Protocol Length Info
1 2022-09-14 04:0… 127.0.0.1 127.0.0.1 TCP 76 39468 → 4433 [SYN] Seq=17
2 2022-09-14 04:0… 127.0.0.1 127.0.0.1 TCP 76 4433 → 39468 [SYN, ACK] S
3 2022-09-14 04:0… 127.0.0.1 127.0.0.1 TCP 68 39468 → 4433 [ACK] Seq=17
4 2022-09-14 04:0… 127.0.0.1 127.0.0.1 TCP 94 4433 → 39468 [PSH, ACK] S
5 2022-09-14 04:0… 127.0.0.1 127.0.0.1 TCP 68 39468 → 4433 [ACK] Seq=17
6 2022-09-14 04:0… 127.0.0.1 127.0.0.1 TCP 68 4433 → 39468 [FIN, ACK] S
7 2022-09-14 04:0… 127.0.0.1 127.0.0.1 TCP 68 39468 → 4433 [FIN, ACK] S
8 2022-09-14 04:0… 127.0.0.1 127.0.0.1 TCP 68 4433 → 39468 [ACK] Seq=56
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 4433, Dst Port: 39468, Seq: 569466345, Ack: 1791101687, Len: 26
Source Port: 4433
Destination Port: 39468
[Stream index: 0]
[TCP Segment Len: 26]
Sequence number: 569466345
[Next sequence number: 569466371]
Acknowledgment number: 1791101687
1000 .... = Header Length: 32 bytes (8)
Flags: 0x018 (PSH, ACK)
Window size value: 512
[Calculated window size: 65536]
[Window size scaling factor: 128]
Checksum: 0xfe42 [unverified]
[Checksum Status: Unverified]
Urgent pointer: 0
Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
[SEQ/ACK analysis]
[Timestamps]
TCP payload (26 bytes)
Data (26 bytes)
Data: 576564205365702031342030343a30383a32322032303232…
[Length: 26]
0000 00 00 03 04 00 06 00 00 00 00 00 00 00 08 00
0010 45 00 00 4e c3 9c 40 00 40 06 79 0b 7f 00 00 01
0020 7f 00 00 01 11 51 9a 2c 21 f1 5d e9 6a c2 0a f7
0030 80 18 02 00 fe 42 00 00 01 01 08 0a 65 ce 7f 67
0040 65 ce 7f 66 57 65 64 20 53 65 70 20 31 34 20 30
0050 34 3a 30 38 3a 32 32 20 32 30 32 32 0d 0a

图 4: 数据包 4