

[illegible]

Wireshark

Capturing from h1-eth0 [Wireshark 1.12.1 (Git Rev Unknown from unknown)]

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help

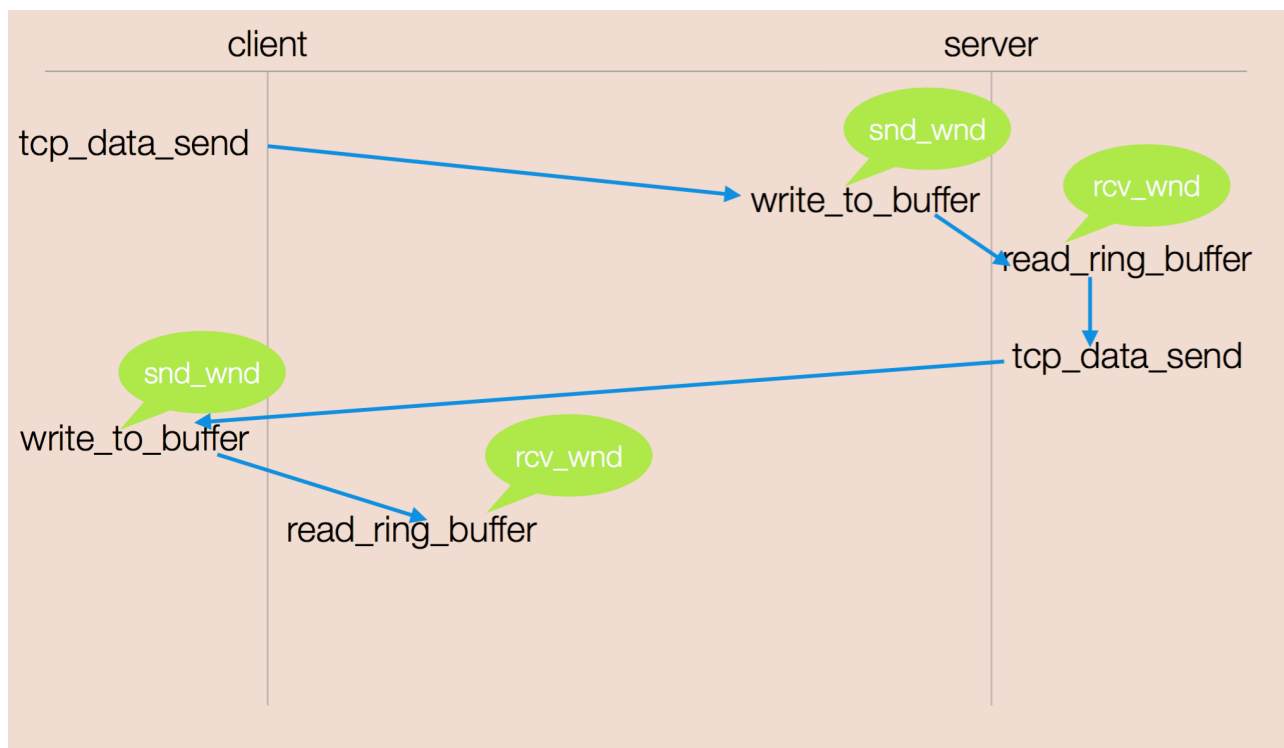
Filter: Expression... Clear Apply Save

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	66:d5:cc:51:34:3e	Broadcast	ARP	42	Who has 10.0.0.1? Tell 10.0.0.2
2	0.000257000	5a:1e:03:06:f1:89	66:d5:cc:51:34:3e	ARP	42	10.0.0.1 is at 5a:1e:03:06:f1:89
3	0.000281000	10.0.0.2	10.0.0.1	TCP	54	12346-10001 [SYN] Seq=0 Win=65535 Len=0
4	0.000556000	10.0.0.1	10.0.0.2	TCP	54	10001-12346 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0
5	0.000615000	10.0.0.2	10.0.0.1	TCP	54	12346-10001 [ACK] Seq=1 Ack=1 Win=65535 Len=0
6	0.001287000	10.0.0.2	10.0.0.1	TCP	116	12346-10001 [PSH, ACK] Seq=1 Ack=1 Win=65535 Len=62
7	0.001808000	10.0.0.1	10.0.0.2	TCP	131	10001-12346 [PSH, ACK] Seq=1 Ack=2 Win=65534 Len=77
8	1.002259000	10.0.0.2	10.0.0.1	TCP	116	12346-10001 [PSH, ACK] Seq=63 Ack=2 Win=65535 Len=62
9	1.003594000	10.0.0.1	10.0.0.2	TCP	131	10001-12346 [PSH, ACK] Seq=78 Ack=64 Win=65534 Len=77
10	2.003619000	10.0.0.2	10.0.0.1	TCP	116	12346-10001 [PSH, ACK] Seq=125 Ack=79 Win=65534 Len=62
11	2.004337000	10.0.0.1	10.0.0.2	TCP	131	10001-12346 [PSH, ACK] Seq=155 Ack=126 Win=65534 Len=77
12	3.004287000	10.0.0.2	10.0.0.1	TCP	116	12346-10001 [PSH, ACK] Seq=187 Ack=156 Win=65534 Len=62
13	3.006183000	10.0.0.1	10.0.0.2	TCP	131	10001-12346 [PSH, ACK] Seq=232 Ack=188 Win=65534 Len=77
14	4.004409000	10.0.0.2	10.0.0.1	TCP	116	12346-10001 [PSH, ACK] Seq=249 Ack=233 Win=65534 Len=62
15	4.005514000	10.0.0.1	10.0.0.2	TCP	131	10001-12346 [PSH, ACK] Seq=309 Ack=250 Win=65534 Len=77
16	5.005064000	10.0.0.2	10.0.0.1	TCP	116	12346-10001 [PSH, ACK] Seq=311 Ack=310 Win=65534 Len=62
17	5.006025000	10.0.0.1	10.0.0.2	TCP	131	10001-12346 [PSH, ACK] Seq=386 Ack=312 Win=65534 Len=77
18	6.005450000	10.0.0.2	10.0.0.1	TCP	116	12346-10001 [PSH, ACK] Seq=373 Ack=387 Win=65534 Len=62
19	6.005785000	10.0.0.1	10.0.0.2	TCP	131	10001-12346 [PSH, ACK] Seq=463 Ack=374 Win=65534 Len=77
20	7.006519000	10.0.0.2	10.0.0.1	TCP	116	12346-10001 [PSH, ACK] Seq=435 Ack=464 Win=65534 Len=62
21	7.007415000	10.0.0.1	10.0.0.2	TCP	131	10001-12346 [PSH, ACK] Seq=540 Ack=436 Win=65534 Len=77
22	8.007395000	10.0.0.2	10.0.0.1	TCP	116	12346-10001 [PSH, ACK] Seq=497 Ack=541 Win=65534 Len=62
23	8.007737000	10.0.0.1	10.0.0.2	TCP	131	10001-12346 [PSH, ACK] Seq=617 Ack=498 Win=65534 Len=77
24	9.007568000	10.0.0.2	10.0.0.1	TCP	116	12346-10001 [PSH, ACK] Seq=559 Ack=618 Win=65534 Len=62
25	9.007857000	10.0.0.1	10.0.0.2	TCP	131	10001-12346 [PSH, ACK] Seq=694 Ack=560 Win=65534 Len=77
26	10.007764000	10.0.0.2	10.0.0.1	TCP	54	12346-10001 [FIN, ACK] Seq=621 Ack=695 Win=65534 Len=0
27	10.008012000	10.0.0.1	10.0.0.2	TCP	54	10001-12346 [ACK] Seq=771 Ack=622 Win=65534 Len=0

3. 实验思路

我把h2向h1发送数据定义为一个循环，当我把这这个循环的思路理清之后，实验就相当清晰了！

如下图：



在上图中，client发送数据包后，

server写入到ring_buffer中，之后从ring_buffer中读出，再echo回client

client收到server的echo后，先写入到ring_buffer中，之后从ring_buffer中读出。

我们可以看到，snd_wnd都是在write_to_buffer调整的（通过对端的rcv_wnd）

rcv_wnd都是在read_ring_buffer处调整的(这时自己的ring_buffer状态有了更新)

4. 关键步骤

本次实验的核心是这两个变量：`snd_wnd` 和 `rcv_wnd`

rev_wnd是由tsk自己决定的，tsk通过调整自己的rcv_wnd来表达自己的接收能力。

snd_wnd却是由对端tsk决定的，对端告诉己端它的接收能力，己端就把自己的snd_wnd设置为对端的rcv_wnd的大小。

那么tsk如何调整自己的接收能力呢？

我使用了老师写好的函数：`static inline int ring_buffer_free(struct ring_buffer *rbuf)`

在每一次read_ring_buffer后更新自己的 `rev_wnd`

```
int tcp_sock_read(struct tcp_sock *tsk, char *buf, int len){
    if(ring_buffer_empty(tsk->rcv_buf)){
        return 0;
    }
    int read_len = read_ring_buffer(tsk->rcv_buf, buf, len);
    tsk->rcv_wnd = ring_buffer_free(tsk->rcv_buf);
    printf("New rcv window size: %d\n", tsk->rcv_wnd);

    return read_len;
}
```

那么tsk如何调整 `snd_wnd` 呢？

就是在Established状态下，如果接收到一个TCP_ACK&TCP_PSH的包，读出其中的 `rcv_wnd`，让 `snd_wnd` 等于 `rcv_wnd`

```
case TCP_ESTABLISHED:
    ...
    if(cb->flags & TCP_PSH){
        tsk->rcv_nxt = cb->seq + 1;
        printf("Received a TCP_PSH data, len: %d\n", cb->pl_len);
        write_ring_buffer(tsk->rcv_buf, cb->payload, cb->pl_len);
        tcp_update_window(tsk, cb);
    }
```

不过在本次实验中，发送数据包的大小分别是62和77，每次循环里都能把ring_buffer里的所有内容读完，所以window_size一直都是满的状态～

5.实验总结

本次实验的核心还是要读懂 `tcp_app.c` 里蕴含的逻辑，后面就比较好做了～

另外要读懂ring_buffer部分的源码，通过 `%` 符号，我知道了这个确实是一个ring😂这部分的API也非常清晰，很方便我们在程序中调用

这应该是我在网络课程的最后一次实验报告，谢谢武老师一个学期的辛苦付出！