网络实验报告

雷正宇 2016K8009909005

实验内容

运行给定网络拓扑(tcp_topo.py)

在节点h1上执行TCP程序

执行脚本(disable_offloading.sh , disable_tcp_rst.sh)

在h1上运行TCP协议栈的服务器模式 (./tcp_stack server 10001)

在节点h2上执行TCP程序

执行脚本(disable_offloading.sh, disable_tcp_rst.sh)

在h2上运行TCP协议栈的客户端模式，连接h1并正确收发数据 (./tcp_stack client 10.0.0.1 10001)

client向server发送数据，server将数据echo给client

使用tcp_stack.py替换其中任意一端，对端都能正确收发数据

修改tcp_apps.c(以及tcp_stack.py)，使之能够收发文件

执行create_randfile.sh，生成待传输数据文件client-input.dat

运行给定网络拓扑(tcp_topo.py)

在节点h1上执行TCP程序

执行脚本(disable_offloading.sh , disable_tcp_rst.sh)

在h1上运行TCP协议栈的服务器模式 (./tcp_stack server 10001)

在节点h2上执行TCP程序

执行脚本(disable_offloading.sh, disable_tcp_rst.sh)

在h2上运行TCP协议栈的客户端模式 (./tcp_stack client 10.0.0.1 10001)

Client发送文件client-input.dat给server，server将收到的数据存储到文件server-output.dat

使用md5sum比较两个文件是否完全相同

使用tcp_stack.py替换其中任意一端，对端都能正确收发数据

实验流程

序号和确认号的设置

本次实验看似内容不多，实则难度很大，一个原因在于它将上一周实验中隐藏的问题暴露了出来，其中一个重要的问题就是序列号和确认号的问题。在本实验环境中，用这样几个字段来指示 tcp 套接字的属性：

iss: 初始发送序号。一条 TCP 连接的双方均可以随机地选择初始序号，这样做可以减少将那些仍在网络中存在的来自两台主机之间先前已终止的连接的报文段误认为是后来这两台主机之间新建连接所产生的有效报文段的可能性。

snd_una: 最后被另一端确认的序号。也就是基序号的前一个序号。

snd_nxt: 下一个发送序号。也就是最小的未使用的序号。

rcv_nxt: 下一个接收序号。也就是最后确认收到的序号的后一个序号。

在接收 TCP 数据包时，针对序号和确认号和窗口大小进行检查；发送数据包时，更新窗口并设置序号和确认号：

```
tcp_update_window_safe(tsk, cb);
tsk->snd_una = cb->ack;
tsk->rcv_nxt = cb->seq_end;
```

接收TCP数据报过程

当套接字处于 established 状态，并收到 ACK 和 PSH 置为1的数据报时，使用 write_ring_buffer 将数据包中的内容复制到环形缓存区：

```
tcp_update_window_safe(tsk, cb);
tsk->snd_una = cb->ack;
tsk->rcv_nxt = cb->seq_end;
if (cb->pl_len > 0) {
  write_ring_buffer(tsk->rcv_buf, cb->payload, cb->pl_len);
  wake_up(tsk->wait_recv);
}
tcp_send_control_packet(tsk, TCP_ACK);
```

当应用层发出读取指令（tcp_sock_read）时，使用 read_ring_buffer 将数据从环形缓存区复制出来：

```
int tcp_sock_read(struct tcp_sock *tsk, char *buf, int len)
{
  while (ring_buffer_empty(tsk->rcv_buf)) {
    sleep_on(tsk->wait_recv);
  }

  int read_len = read_ring_buffer(tsk->rcv_buf, buf, len);
  tsk->rcv_wnd = ring_buffer_free(tsk->rcv_buf);

  return read_len;
}
```

该函数返回值为最终读取的字节数。读取内容后要根据接收缓存大小更新窗口大小。

发送TCP数据报过程

同样，当套接字处于 established 状态，应用层发出发送指令时（tcp_sock_write），将数据封装并发送：

```
int tcp_sock_write(struct tcp_sock *tsk, char *buf, int len)
{
  while (tsk->snd_wnd == 0) {
    sleep_on(tsk->wait_send);
  }

  int head_len = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + TCP_BASE_HDR_SIZE;
  int rest_len = len + head_len;
  int packet_len = min(ETH_FRAME_LEN, rest_len);
  char *packet = malloc(packet_len);
  memcpy(packet + head_len, buf, packet_len - head_len);

  tcp_send_packet(tsk, packet, packet_len);

  return packet_len - head_len;
}
```

该函数返回值为最终发送的字节数（不包括头部）。这里有一个要注意的地方，就是如果用这种简单的方式处理 write 过程，就必须保证调用者在调用函数之前确定接收方的窗口足够大。

收发文件的应用

为了确认功能的正确性，需要自己编写一个能收发文件的应用。服务器端将收到的信息转化为字节流写入一个文件，用户端将一个本地文件发送到服务器。python 代码如下：

```
def server(port):
    s = socket.socket()
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    s.bind(('0.0.0.0', int(port)))
    s.listen(3)

    while True:
        cs, addr = s.accept()
        print(addr)
        file = open(output_data_name, mode='w+')
        data = cs.recv(1000)
        while data:
            file.write(data.decode('utf-8'))
            data = cs.recv(1000)

        file.close()
```

```
        cs.close()

    s.close()

def client(ip, port):
    s = socket.socket()
    s.connect((ip, int(port)))
    file = open(input_data_name, 'r')

    buf = file.read(500)
    while buf:
        s.send(buf.encode('utf-8'))
        buf = file.read(500)

    file.close()
    s.close()
```

C 语言版本的应用要更复杂一些：

```
void *tcp_server(void *arg)
{
  u16 port = *(u16 *)arg;
  struct tcp_sock *tsk = alloc_tcp_sock();

  struct sock_addr addr;
  addr.ip = htonl(0);
  addr.port = port;
  if (tcp_sock_bind(tsk, &addr) < 0) {
    log(ERROR, "tcp_sock bind to port %hu failed", ntohs(port));
    exit(1);
  }

  if (tcp_sock_listen(tsk, 3) < 0) {
    log(ERROR, "tcp_sock listen failed");
    exit(1);
  }

  log(DEBUG, "listen to port %hu.", ntohs(port));
  struct tcp_sock *csk = tcp_sock_accept(tsk);
  log(DEBUG, "accept a connection.");

  char rbuf[1001];
  int rlen = 0;
  fopen("./server-output.dat", "w");
  while (true) {
    rlen = tcp_sock_read(csk, rbuf, 500);
    if (rlen == 0) {
      log(DEBUG, "tcp_sock_read return 0, finish transmission.");
      break;
```

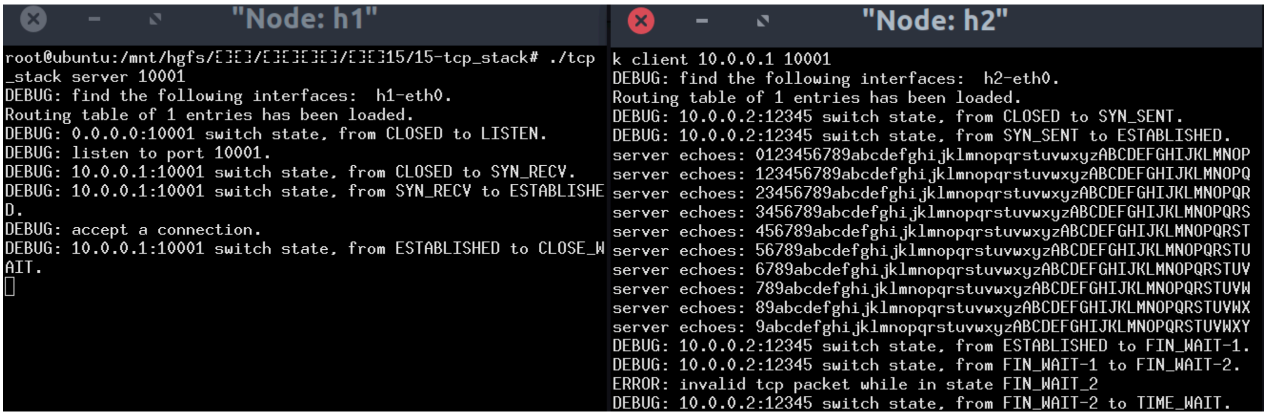
```
    }
    else if (rlen > 0) {
      FILE *file = fopen("./server-output.dat", "a");
      printf("wlen: %ld\n", fwrite(rbuf, rlen, 1, file));
      fclose(file);
    }
    else {
      log(DEBUG, "tcp_sock_read return negative value, something goes
wrong.");
      exit(1);
    }
  }

  log(DEBUG, "close this connection.");

  tcp_sock_close(csk);

  return NULL;
}

void *tcp_client(void *arg)
{
  struct sock_addr *skaddr = arg;
  struct tcp_sock *tsk = alloc_tcp_sock();
  if (tcp_sock_connect(tsk, skaddr) < 0) {
    log(ERROR, "tcp_sock connect to server ("IP_FMT":%hu)failed.", \
        NET_IP_FMT_STR(skaddr->ip), ntohs(skaddr->port));
    exit(1);
  }

  FILE *file = fopen("./client-input.dat", "r");

  char rbuf[1001];
  int rlen = fread(rbuf, 1, 1000, file);

  while (rlen > 0) {
    printf("rlen: %d\n", rlen);
    tcp_sock_write(tsk, rbuf, rlen);
    rlen = fread(rbuf, 1, 1000, file);
  }
  tcp_sock_write(tsk, rbuf, 0);

  fclose(file);

  tcp_sock_close(tsk);

  return NULL;
}
```

C 语言版的应用直接调用在前面编辑的套接字接口，由于一些设计上的问题，服务器应用在运行的同时会和套接字进程发生冲突：服务器应用在 tcp_sock_read 中发生阻塞后，套接字仍然在和另一端（用户进程）通信，并可能改变自己的状态。当用户向服务器发送有效报文后，服务器应用还没有正确读取缓存区内的数据，套接字就有可能离开了 established 状态。为了避免这种情况发生， tcp_sock_read 和 tcp_sock_write 需要进行修改。但目前我还没有来得及针对这种情况进行完善的改动，所以我暂时不判断进程从 read 队列中被唤醒时套接字的状态。

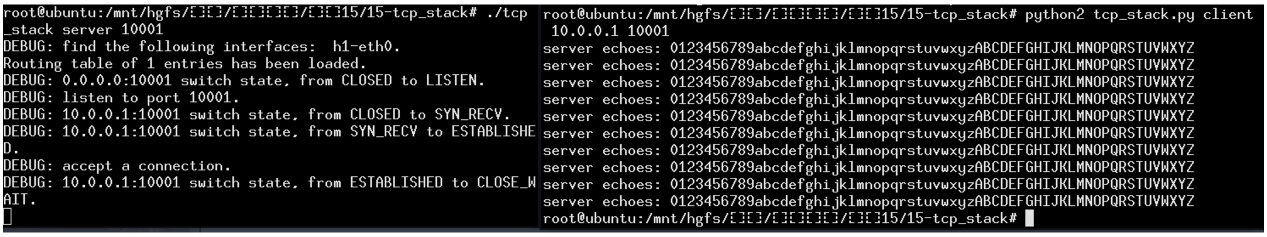
这样的临时处理导致的另一个后果是服务器程序无法通过判断 rlen == 0 来正常退出（C 语言对异常处理控制流的支持不是很好），从而使得套接字和文件的关闭等操作不能正常进行。对此，我利用文件的追加写入功能，暂且避免了文件写入失败的问题。

实验结果和分析

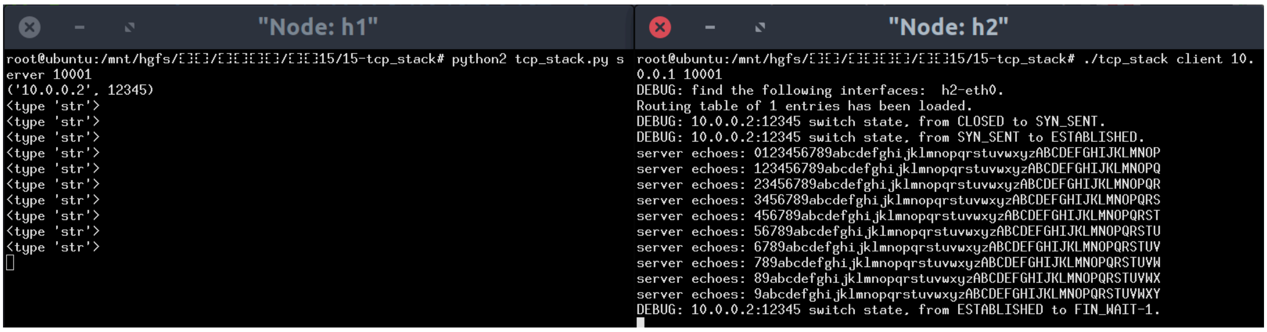
首先是完成基本 API 编写后，echo 应用的运行结果：



下面是将 client 端换为 python 脚本的运行结果：



下面是将 server 端换为 python 脚本的运行结果：



最后是传输文件的实验。这部分通过截图难以体现正确性，暂且用文字描述。

基本过程是先使用 shell 脚本创建一个 ASCII 文件 input，启动两个 tcp 应用，用户每次读取 input 文件 1000 字节并发送给服务器，服务器每次读取 500 字节并写入 output 文件。然后通过 md5sum 或者 diff 工具验证两个文件是否完全相同。

在文件大小不超过 50KB 的情况下，文件传输实验是成功的（使用 diff 确认两个文件完全相同）。但是当文件超过 50KB 后，且服务器应用使用我自己编写的协议栈时，就有发生崩溃的可能。这个问题在实验提交时仍然没有成功解决，只能期待在下一周的实验中进行解决了。