实验 4 TCP 协议实现

本次实验选用 rust sdk 完成，只改动了代码文件`outgoing.rs`和配置文件`Cargo.toml`，编译得到的可执行文件见`bin/network_exp4_sdk`。

1 实验思路

1.1 状态存储

为实现有状态的 TCP 协议，程序为每条 TCP 连接定义了保存状态的数据结构：

```rust
struct TCPState {
    // 存储一条 TCP 连接状态的 struct
    // state: tcp 连接当前在 FSM 图中的位置
    // seq: 下一个发送的 tcp 报文序号数
    // ack: 发送的 ack 序号数
    // acked: 未被确认的最小序号数
    // dup_acks: 重复 ack 个数,超过三次快速重传.
    // EstRTT: RTT 估计值
    // DevRTT: RTT 偏差估计值
    // send_time: 每条报文的发送时间, key 为报文序列号
    // send_times2: 每条报文发送时间, key 为报文到达后对端预期返回的 ack 号, 这是为了方便计算 estRTT
    // send_cache: 缓存发送过的报文以备超时重传, key 为报文序列号
    state: FSMState,
    seq: u32,
    ack: u32,
    acked: u32,
    dup_acks: usize,
    EstRTT: time::Duration,
    DevRTT: time::Duration,
    send_times: HashMap<u32, time::Instant>,
    send_times2: HashMap<u32, time::Instant>,
    send_cache: HashMap<u32, TCP_packet>,
}
```

其中参考 TCP 状态图 定义 TCP 发起端所需的 FSM 状态如下：

```rust
enum FSMState {
    SYNSENT,
    ESTABLISHED,
    FINWAIT1,
    FINWAIT2,
    TIMEWAIT,
}
```

利用`HashMap`数据结构保存各条 TCP 连接的状态：

```rust
lazy_static! {
    // 存储所有在线 TCP 连接状态, key 为 &ConnectionIdentifier2Str(&conn)
    static ref TCPSTATES:Mutex<HashMap<String, TCPState>> = Mutex::new(HashMap::new());
    // 存储所有在线 TCP 连接四元组
    static ref CONNECTIONS:Mutex<Vec<ConnectionIdentifier>> = Mutex::new(Vec::new());
}
```

每当发送/接收 TCP 报文时，程序读取或修改对应 TCP 连接的状态变量以构造、发送、解析对应的 TCP 报文，完成与对端的通信，各函数的实现见源代码及注释。

1.2 超时重传和快速重传

程序实现了**超时重传**和**快速重传**机制。

其中超时重传机制在`tick`函数中实现，每次`tick()`被调用，程序会遍历目前在线的所有 TCP 连接，检查每条 TCP 连接已发送的最小序号未确认报文是否超时，如果超时，重传报文。

根据 TCP 标准，超时阈值设置为`EstRTT + 4DevRTT`，每次收到对端的 ACK 报文时，程序从 TCP 状态变量中读取对应报文的发送时间，计算出`SampleRTT`，并用以下滑动平均公式更新该 TCP 连接`EstRTT`和`DevRTT`变量的值：

$$\mathrm{EstRTT} = (1-\alpha)\cdot \mathrm{EstRTT} + \alpha \cdot \mathrm{SampleRTT}$$
$$\mathrm{DevRTT} = (1-\beta)\cdot \mathrm{DevRTT} + \beta \cdot |\mathrm{SampleRTT} - \mathrm{EstRTT}|$$

其中α取推荐值0.125，β取推荐值0.25。

此外，程序会在`TCPState.dup_acks`中记录 TCP 连接收到的重复 ACK 数目，如果收到三个重复 ACK 则重传相应报文，实现快速重传。

2 测试结果

2.1 基础功能测试

运行`curl baidu.com`，运行结果：

```
$ curl baidu.com
<html>
<meta http-equiv="refresh" content="0;url=http://www.baidu.com/">
</html>
```

抓包得到相关 TCP 报文如下（wireshark抓包截图）：

```
1 0.000000000 172.24.248.2 → 39.156.66.10 TCP 54 63744 → 80 [SYN] Seq=0 Win=1460 Len=0
2 0.008792060 39.156.66.10 → 172.24.248.2 TCP 60 80 → 63744 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0
3 0.009130742 172.24.248.2 → 39.156.66.10 TCP 54 63744 → 80 [ACK] Seq=1 Ack=1 Win=1460 Len=0
4 0.051279901 172.24.248.2 → 39.156.66.10 HTTP 127 GET / HTTP/1.1
5 0.061402087 39.156.66.10 → 172.24.248.2 TCP 60 80 → 63744 [ACK] Seq=1 Ack=74 Win=24684 Len=0
6 0.061690190 39.156.66.10 → 172.24.248.2 TCP 359 HTTP/1.1 200 OK  [TCP segment of a reassembled PDU]
7 0.061690276 39.156.66.10 → 172.24.248.2 HTTP 135 HTTP/1.1 200 OK  (text/html)
8 0.062273966 172.24.248.2 → 39.156.66.10 TCP 54 63744 → 80 [ACK] Seq=74 Ack=306 Win=1460 Len=0
9 0.062460305 172.24.248.2 → 39.156.66.10 TCP 54 63744 → 80 [ACK] Seq=74 Ack=387 Win=1460 Len=0
10 0.062932126 172.24.248.2 → 39.156.66.10 TCP 54 63744 → 80 [FIN] Seq=74 Win=1460 Len=0
11 0.073086023 39.156.66.10 → 172.24.248.2 TCP 60 80 → 63744 [ACK] Seq=387 Ack=75 Win=24684 Len=0
12 0.073086113 39.156.66.10 → 172.24.248.2 TCP 60 80 → 63744 [FIN, ACK] Seq=387 Ack=75 Win=24684 Len=0
13 0.073722935 172.24.248.2 → 39.156.66.10 TCP 54 63744 → 80 [ACK] Seq=75 Ack=388 Win=1460 Len=0
```

可见：

- 程序正确地进行了 TCP 协议的**三次握手**以建立连接
- 程序正确地处理了应用层 HTTP GET 的发送报文和对端的 HTTP 响应
- 程序正确进行了 TCP 协议的**四次挥手**以关闭连接
- 程序能够双向收发小规模数据

2.2 接收数据测试

```
$ curl https://mirrors.bfsu.edu.cn/ubuntu/pool/main/p/python3.10/libpython3.10-dev_3.10.9-1_amd64.deb -o libpython3.10-dev_3.10.9-1_amd64.deb
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 4541k  100 4541k    0     0  77764      0  0:00:59  0:00:59 --:--:--  102k
```

检查文件 Hash 值：

```
$ sha256sum libpython3.10-dev_3.10.9-1_amd64.deb
6b803051e551a708f74e2fc6823797b6076e94637c758fbdfd0488891b7d4551  libpython3.10-dev_3.10.9-1_amd64.deb
```

与参考值相同。

2.3 发送数据测试

```
$ dd if=/dev/urandom of=/tmp/testfile bs=1M count=3
curl -vvv http://lab.starrah.cn:28020/test_upload -F "file=@/tmp/testfile"
```

得到文件哈希值与本地`sha256sum`得到的哈希值相同。

> 测试过程中发现上传文件速度较慢，猜测是因为程序没有实现流量控制，会将应用层发来的数据立刻发出，使对端缓存溢出导致丢包需要重传，而`tick`函数约 `100ms` 才会被调用一次，每次只会重传一个包，导致上传缓慢。

2.4 超时重传

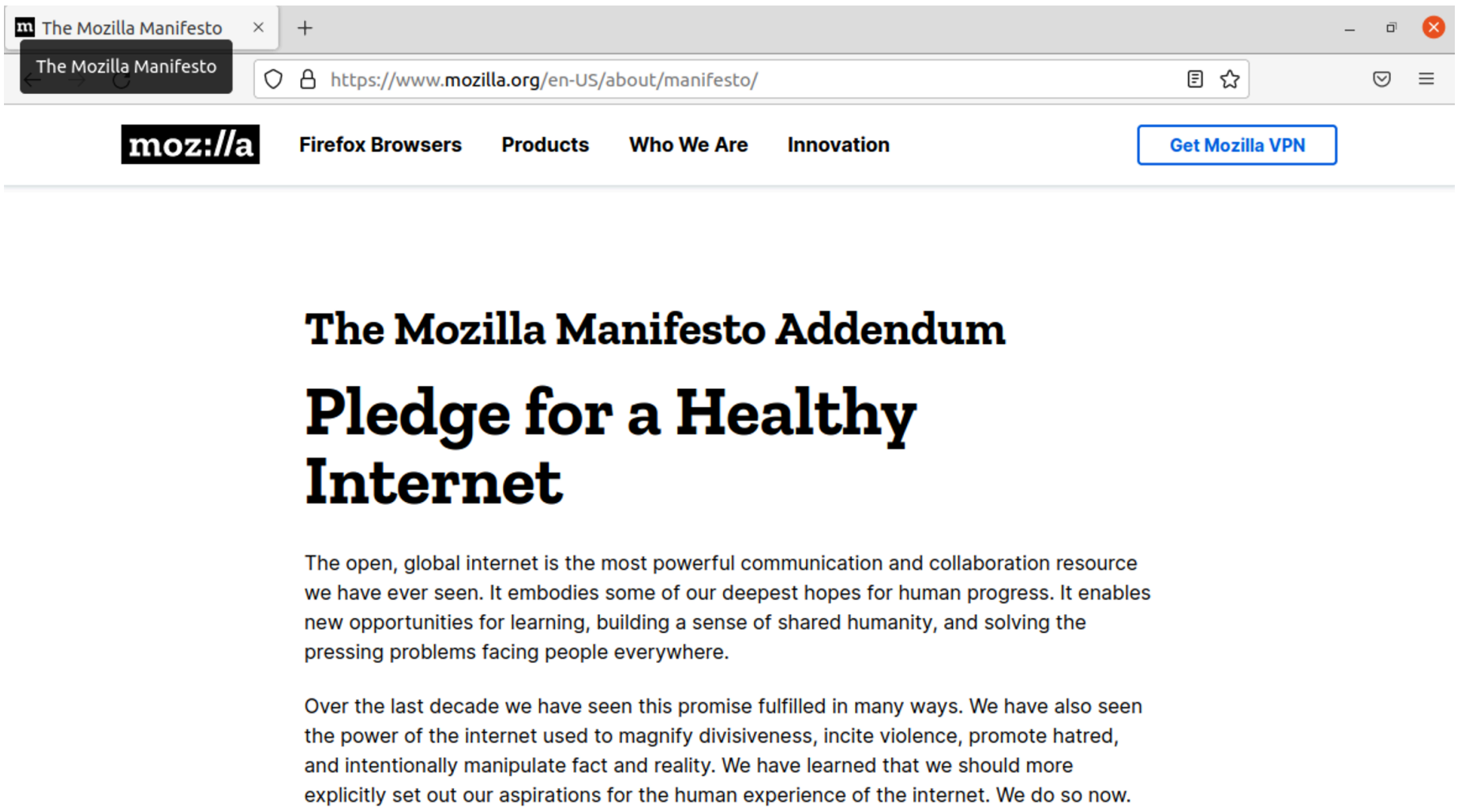
由于未实现流量控制，测试上传过程中容易发生丢包，触发超时重传机制，如：

3338	1.272087556	166.111.83.24	172.24.248.2	TCP	60	28020 → 61248 [ACK] Seq=26 Ack=144150 Win=65535 Len=0
3339	1.365520527	172.24.248.2	166.111.83.24	TCP	134	[TCP Retransmission] 61248 → 28020 [ACK] Seq=144150 Ack=26 Win=1460 Len=80
3340	1.372198531	166.111.83.24	172.24.248.2	TCP	60	28020 → 61248 [ACK] Seq=26 Ack=144230 Win=65535 Len=0
3341	1.467451865	172.24.248.2	166.111.83.24	TCP	1514	[TCP Retransmission] 61248 → 28020 [ACK] Seq=144230 Ack=26 Win=1460 Len=1460
3342	1.474146324	166.111.83.24	172.24.248.2	TCP	60	28020 → 61248 [ACK] Seq=26 Ack=145690 Win=65535 Len=0
3343	1.569347780	172.24.248.2	166.111.83.24	TCP	1514	[TCP Retransmission] 61248 → 28020 [ACK] Seq=145690 Ack=26 Win=1460 Len=1460
3344	1.576226778	166.111.83.24	172.24.248.2	TCP	60	28020 → 61248 [ACK] Seq=26 Ack=147150 Win=65535 Len=0
3345	1.669351286	172.24.248.2	166.111.83.24	TCP	134	[TCP Retransmission] 61248 → 28020 [ACK] Seq=147150 Ack=26 Win=1460 Len=80
3346	1.676131438	166.111.83.24	172.24.248.2	TCP	60	28020 → 61248 [ACK] Seq=26 Ack=147230 Win=65535 Len=0
3347	1.770460289	172.24.248.2	166.111.83.24	TCP	1514	[TCP Retransmission] 61248 → 28020 [ACK] Seq=147230 Ack=26 Win=1460 Len=1460
3348	1.777224149	166.111.83.24	172.24.248.2	TCP	60	28020 → 61248 [ACK] Seq=26 Ack=148690 Win=65535 Len=0

图中上传过程中缓存溢出导致丢包，触发了每 100ms 调用的`tick`函数内超时重传机制，从对方返回的 ACK 报文可见超时重传机制正确运行。

2.5 浏览网页

经测试程序可以打开网页：



3 实验收获

通过此次实验熟悉了TCP协议的实现细节，加深了我对传输层协议在整个OSI模型中发挥作用的理解。

实际编写程序的过程中遇到了很多 bug，调试的过程也锻炼了我使用`wireshark`等网络工具的能力以及用`rust`编写底层网络程序的能力，最后感谢助教帮助我排查解决了一些问题。