# TCP可靠传输实验报告

# 项目成员

---

- 黄成，学号201828013229149。

# 实验内容

---

0. 配置实验环境：安装mininet和相关工具。
1. 实现TCP数据传输功能：在给定的网络拓扑下，建立Client-Server的TCP连接，实现无丢包环境中的数据传输功能，最后关闭连接。
2. 在1的基础上，实现TCP的可靠传输功能：在有丢包的网络环境中，通过超时重传机制实现恢复丢包的可靠传输功能。
3. 在2的基础上，实现TCP的拥塞控制功能：实现类似TCP-newReno中的慢启动、快速重传、快速恢复等拥塞控制功能。

# 实验流程

---

## 1.安装Linux虚拟机、mininet和相关工具

略。

## 2.TCP数据传输

### 项目代码梳理

本实验项目代码中涉及的文件较多，主要有以下几个部分：

- 一些基础的.h文件，如日志、类型、monitor、hash函数、环形缓冲区(ring buffer)、双向队列等相关的定义和操作；
- 网络协议栈相关的基础的.h文件，如校验和、网口(interface)、路由表、数据包等相关的定义和操作，以及对应的.c文件；
- 一些具体的网络协议相关的.h文件，如ARP, ICMP, IP, TCP等，以及对应的.c文件；
- 与TCP相关的一些细节，如计时器、socket、hash表和相关函数、TCP包的发送和接收，以及定义了server和client各自行为的文件；
- main.c：初始化协议栈，传递参数启动server和client，启动协议栈负责数据包在不同节点间的传输；
- 生成数据、生成网络拓扑等脚本，具体实现底层协议栈的静态库，以及Makefile等文件。

在实验给定的两节点拓扑下，两个节点分别运行TCP server程序和TCP client程序。具体流程如下：

1. server监听固定端口(port)，client根据server的IP地址和端口主动连接，双方进行三次握手；
2. client从输入文件中读取数据，发送给server，server回复对应ACK；
3. client发送结束整个文件后，主动关闭连接，进行前两次挥手；
4. server接收完毕后，也发起结束请求，进行后两次挥手。

### `tcp_sock` 相关接口实现

TCP连接由socket建立和管理，本次实验中需要实现具有类似功能的 `tcp_sock` 的重要接口。

**建立连接**:

`tcp_sock_listen`: 将server端新创建的 `tcp_sock` 设置为 `TCP_LISTEN` 状态，映射到端口(port)hash表和监听hash表；

`tcp_sock_connect`: 设置client端 `tcp_sock`：

1. 设置四元组：根据输入参数设置目标IP地址和端口号，从转发表中查找与目标IP匹配的网口(interface)的IP地址作为源IP地址，获取一个可用的源端口号；
2. 初始化序列号和一些队列；
3. 设置状态为 `TCP_SYN_SENT`，映射到端口hash表和连接(established)hash表；
4. 发送第一次握手的数据包，`sleep_on` 等待连接建立。

`tcp_sock_lookup_established` / `tcp_sock_lookup_listen`: （接收方收到数据包后，需要依次查找连接hash表和监听hash表，寻找与数据包目的端口匹配的 `tcp_sock`，端口相当于上层应用程序的标识符）通过hash后扫描对应bucket，找到符合条件的 `tcp_sock`，否则返回 `NULL`；

`tcp_sock_accept`: （server端的第一个 `tcp_sock` 一直保持监听状态，新的client接入时，需要创建新的child `tcp_sock` 来服务连接）`sleep_on` 等待完成连接建立的child `tcp_sock` 被放入 `accept_queue`，从中取出它并返回；后续server端的所有操作只与这个child `tcp_sock` 相关。

**数据传输**:

`tcp_sock_write`: （client端每次将输入文件读入buffer，然后调用 `tcp_sock_write`）检查发送窗口大小，如果有剩余空间则新建数据包，拷贝数据并发送；否则 `sleep_on` 等待发送窗口出现空余；

`tcp_sock_read`: 检查ring buffer，如果有数据则读出（需要加锁避免读写冲突），转移到上层buffer中等待输出；否则 `sleep_on` 等待ring buffer中被写入数据。

**关闭连接**:

`tcp_sock_close`: 检查 `tcp_sock` 状态，

- 如果是 `TCP_ESTABLISHED`，说明是client发送完毕主动关闭连接，则发送 `FIN` 包（第一次挥手），状态切换为 `TCP_FIN_WAIT_1`；
- 如果是 `TCP_CLOSE_WAIT`，说明是server接收完毕关闭连接，则发送 `FIN` 包（第三次挥手），状态切换为 `TCP_LAST_ACK`。

## 计时器相关函数实现

在主动关闭连接一方收到对方发送的 `FIN` 包时，需要回复 `ACK` 并设置计时器，如果计时器结束前没有收到任何消息，则代表对方已经正常关闭连接，自己也可以关闭连接：

`tcp_set_timewait_timer`: 设置计时器类型和超时时间，启用计时器，加入 `timer_list`；

`tcp_scan_timer_list`: 一个线程每隔一段时间扫描一次 `timer_list`，每次扫描时启用的计时器超时时间减少，如果timewait计时器超时时间 ≤ 0，删除该计时器，解除对应 `tcp_sock` 相关的hash表映射，释放 `tcp_sock` 空间，整个连接彻底结束。

## 数据包处理流程实现

底层协议栈接收并交付的数据包最终交由 `tcp_process` 函数处理，根据 `flags` 域和 `tcp_sock` 状态的不同有不同的处理流程：

**建立连接**:

TCP_SYN: server端的parent tcp_sock 处于 TCP_LISTEN 状态时接收到，申请一个新的child tcp_sock，初始化四元组、序列号、一些队列、接收窗口，设置状态为 TCP_SYN_RECV，映射到连接hash表（之后的数据包由它接收处理），回复 SYN 和 ACK（第二次握手）；

TCP_SYN | TCP_ACK: client端设置 tcp_sock 发送窗口大小，状态修改为 TCP_ESTABLISHED，回复 ACK（第三次握手），唤醒 tcp_sock_connect 中等待连接建立的线程（该线程开始读入输入数据准备发送）；

TCP_ACK: server端将child tcp_sock 加入 accept_queue 中，设置状态为 TCP_ESTABLISHED，唤醒 tcp_sock_accept 中等待的线程。

**数据传输**：处于 TCP_ESTABLISHED 状态

TCP_PSH | TCP_ACK: 普通数据包，server端检查序列号，加锁并将数据写入ring buffer中，尝试唤醒可能在 tcp_sock_read 中等待读取ring buffer的线程；

TCP_ACK: client端根据新确认的数据和接收窗口大小等信息滑动发送窗口，尝试唤醒 tcp_sock_write 中等待空余窗口的线程。

**关闭连接**：

TCP_FIN:

- 如果 tcp_sock 处于 TCP_ESTABLISH 状态，说明是server端，则将状态设置为 TCP_CLOSE_WAIT，回复 ACK（第二次挥手），并尝试唤醒 tcp_sock_read 中等待读取数据的线程（该线程返回0时将触发server端的关闭）；
- 如果处于 TCP_FIN_WAIT_2 状态，说明是client端，则将状态设置为 TCP_TIME_WAIT，回复 ACK（第四次挥手），启动timewait计时器等待超时关闭；

TCP_ACK:

- 如果 tcp_sock 处于 TCP_FIN_WAIT_1 状态，说明是client端并且已经开始主动关闭连接，则修改状态为 TCP_FIN_WAIT_2；
- 如果处于 TCP_LAST_ACK 状态，说明是server端并且已经开始关闭连接，则修改状态为 TCP_CLOSED，解除hash表映射，释放 tcp_sock；

**其它**：

TCP_RST: 直接将 tcp_sock 状态切换为 TCP_CLOSED，清除计时器、队列、映射，释放 tcp_sock。

# 3.TCP可靠传输

在实际情况下，网络中可能会发生丢包，TCP的可靠传输功能需要完成丢包的检测和恢复功能，主要依赖于超时重传机制（需要对应增加 send_buf, rcv_ofo_buf 的初始化）：

tcp_set_retrans_timer / tcp_unset_retrans_timer: 开启 / 关闭重传计时器，实现方法与 tcp_set_timewait_timer 类似；

tcp_scan_timer_list: 重传计时器 ≤ 0时，重新发送对应 tcp_sock 的 send_buf 中的第一个数据包，增加该数据包的重传次数计数，超时时间翻倍；重传超过3次后直接关闭连接；

tcp_send_packet / tcp_send_control_packet: 每次发送数据包前，将（除 ACK 包外的）数据包加入 send_buf 末尾，如果加入前 send_buf 为空，则启动重传计时器；

tcp_process **接收到** TCP_ACK **包**: 将新确认的数据从 send_buf 中删除，如果删除后 send_buf 为空则关闭重传计时器，否则重置计时器；

tcp_process **接收到普通数据包**:

- 如果与已经确认的数据包序列号不连续，则放入 rcv_ofo_buf 中；
- 如果连续，则写入ring buffer，并检查 rcv_ofo_buf 中的数据，将所有连续的数据都写入ring buffer。
- 需要注意，client端主动关闭的 FIN 包也可能需要被缓存到 rcv_ofo_buf 中，当它被顺序取出时应按照正常的接收 FIN 包的流程处理。

由于所有的 ACK 包都不会参与重传，对于丢失的 ACK 包需要进行特别的处理（尽管理论上未收到 ACK 的一方可以仅通过超时重传解决这个问题，但这意味着大量的等待）：

- 建立连接时，第三次握手的 ACK 包使server端的状态从 TCP_SYN_RCV 转换为 TCP_ESTABLISHED，如果它丢失，server端可能在 TCP_SYN_RCV 状态下收到第一个数据包，直接转换状态并回复数据包的 ACK 即可；
- 普通数据包的 ACK 信息已经包含在序列号更大数据包的 ACK 中，因此不影响可靠传输；
- 断开连接时，第二次挥手的 ACK 包使client端状态从 TCP_FIN_WAIT_1 转换为 TCP_FIN_WAIT_2，如果它丢失，client端在收到第三次挥手的 FIN 包时直接进入 TCP_TIME_WAIT 并发送 ACK（第四次挥手）即可；
- 如果第四次挥手的 ACK 丢失，通过超时重传解决（server重新发送第三次挥手的 FIN 包）。

## 4.TCP拥塞控制

拥塞控制的核心思想是通过 ACK 的信息提前判断网络中的拥塞和丢包情况，调整发送窗口大小（发送速率）避免过多丢包。需要增加对相关窗口、ssthresh 等变量的初始化。

tcp_process **接收到普通数据包**: 如果数据包乱序（可能由网络拥塞导致），则发送重复的 ACK；

tcp_process **接收到** TCP_ACK **包**:

- 如果在普通状态（非快速恢复）下确认了新数据，根据慢启动或拥塞避免策略（由 cwnd 与 ssthresh 确定）更新拥塞窗口 cwnd；
- 如果收到连续3个重复 ACK，进入快速重传阶段，调整 ssthresh 和 cwnd，设置当前已发送的最新数据包为恢复点，进入快速恢复阶段，重新发送目前尚未确认的序列号最小的数据包；
- 如果在快速恢复状态下确认了新数据而且尚未达到恢复点，重新发送目前尚未确认的序列号最小的数据包；
- 如果在快速恢复状态下确认了新数据并且达到了恢复点，重新进入普通状态。

tcp_scan_timer_list: 如果在拥塞控制下仍然发生了超时重传，需要进一步调整 cwnd 和 ssthresh 的大小，控制发送速率。

# 实验结果及分析

---

由于拥塞控制需要以前两部分为基础进行实现，而且前两部分除了简单的“通过”之外没有更好的结果呈现方法，因此只对拥塞控制部分的实验结果进行分析。

如图为拥塞窗口 cwnd 和慢启动门限值 ssthresh 随时间变化的示意，可以看出两条曲线基本符合TCP newReno的拥塞避免策略：

- 每次快速重传触发，拥塞窗口大小减半，慢启动门限值；
- 由于每次快速恢复时重传丢包会导致连续确认，因此 cwnd 在短时间内会有较快的增长；
- 在10000ms左右触发了一次超时重传，cwnd 减小为1MSS，进入慢启动阶段。

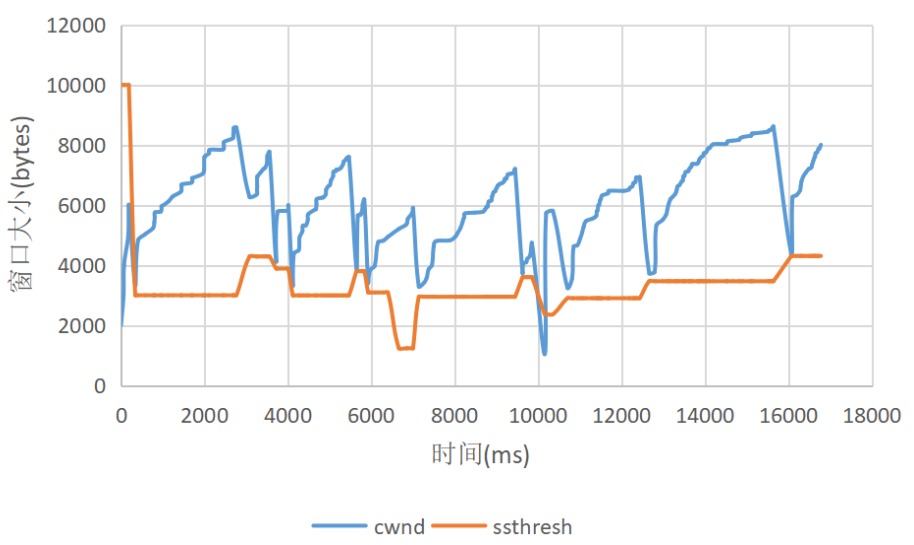


图1 拥塞控制相关窗口大小变化示意。

# 其它相关事项

1. 序列号环绕：根据 `tcp_update_window_safe` 函数提供的方法，先将无符号数转化为有符号数，再相减比较：（假设被减数代表更大的数据包序列号）
    - 当发生整数环绕时，实际上是一个小的正数减一个小的负数，结果为正；
    - 当两个序列号转化后正好跨过正负边界，实际上是一个大的负数减一个大的正数，结果溢出，但符号位为正，因此结果为正。因此解决了整数环绕问题。

2. 由于client每次读取文件的buffer大小只有1000 bytes，发送的数据包不可能写满一整个以太网帧，因此拥塞避免中MSS直接设置为1000 bytes；
3. 新增数据结构 `pending_pkt`，用于将数据包加入 `send_buf`；`ofo_pkt`，用于将收到乱序数据包加入 `rcv_ofo_buf`；
4. 关于选题和实验：出于实验项目代码本身耦合较强以及锻炼个人能力两个原因，我选择了不与其他同学组队完成实验。实际上在完成时间和代码细节的完善性上都不尽如人意（在加入拥塞控制之后似乎引入了新的bug，会出现三次超时导致连接关闭的情况）。但由于临近期末，时间精力不允许，我决定不再对代码细节进行进一步修正。总体而言这次实验让我有些头疼，但同时也有很大的收获，对基本的TCP协议细节有更多的了解；
5. 建议完善代码中的 TODO 标识，例如提供类似3中的必要的数据结构等，使得逻辑上只用实现标有 TODO 的部分就能达到预期的实验结果。