网络传输机制实验一

武庆华

wuqinghua@ict.ac.cn

主要内容

- 网络传输机制 实验一（无丢包环境）
  - TCP理论知识回顾
  - Socket数据结构
  - 连接管理&数据包处理
  - 数据发送与接收
  - TCP协议栈实现
  - 实验内容

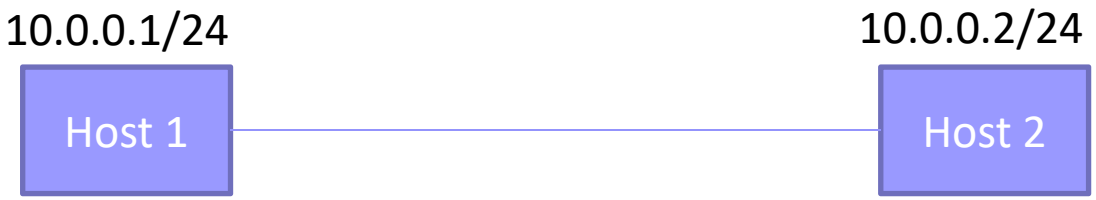
网络传输协议

- 网络层提供了端到端的连接功能
  - 无连接的、尽最大努力交付（best-effort delivery）的数据报服务
- 为了支持网络应用间的数据传输，主机端还需要实现很多功能

- 多路复用
- 连接管理
- 按序传输
- 丢包检测与恢复
- 拥塞控制

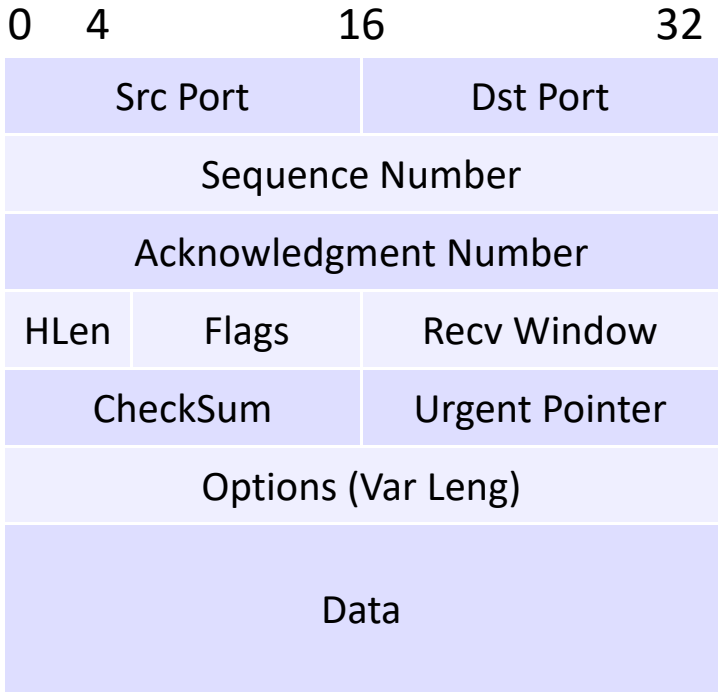
网络传输机制实验

- 网络传输机制实验目的
  1. 连接管理：实现最基本的TCP连接管理功能，使得节点之间能够在无丢包网络环境中建立和断开连接
  2. 数据传输：实现最基本的TCP数据收发功能，使得节点之间能够在无丢包网络环境中传输数据



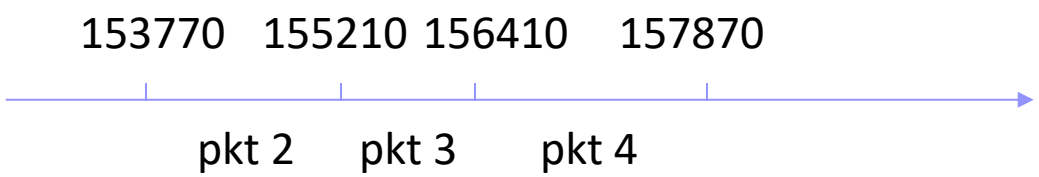
TCP (Transport Control Protocol)

- 应用最广泛的传输层协议
  - 占据了目前互联网的90%以上流量
- 多路复用
  - Src Port, Dst Port
- 连接管理
  - Flags + Seq + Ack
- 可靠传输
  - Seq + Ack + Options
- 流控
  - Recv Window + Options
- 拥塞控制



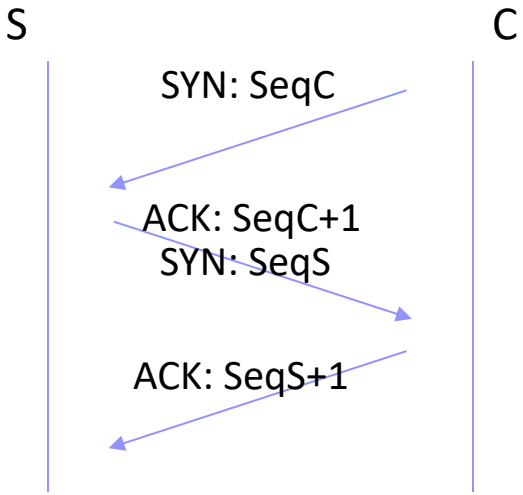
TCP序列号

- TCP是一种面向字节流的传输协议
  - 每个传输字节都对应一个32位整数序列号
    - 当数据大于32位表示时，整数进行环绕
  - 连接建立时用一随机数作为初始序列号
    - 否则容易产生安全性问题
- TCP将数据分割到不同的数据包
  - 数据包中的数据不多于 (1500 - IPHDR - TCPHDR)
  - 每个数据包的序列号是其包含的第一个字节对应的序列号



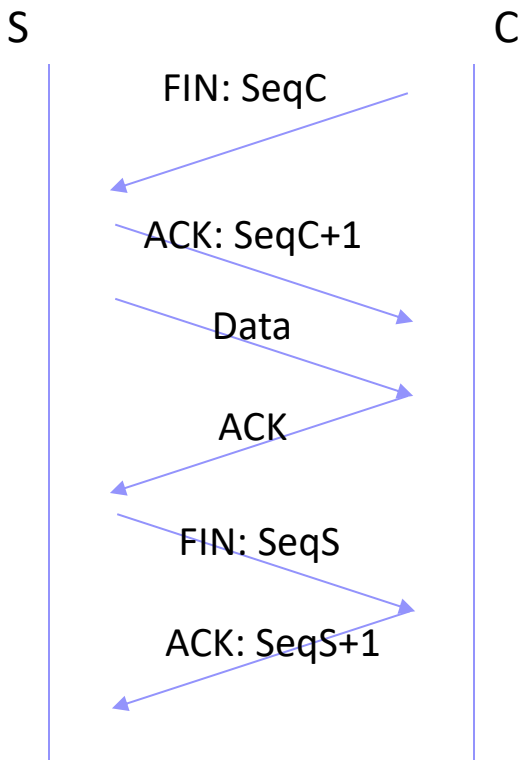
TCP连接建立

- 双方确定建立连接用的端口号(port)
  - 被动建立连接的一方使用固定端口
    - 例如，SSH: 22, HTTP: 80, HTTPS: 443
  - 主动建立连接的一方使用随机端口
    - 由协议栈确定
- 连接双方确定自己的初始序列号并通知对方 (SYN)
  - 两端的初始序列号相互独立
- 双方在收到对方的初始序列号后，回复确认 (ACK)
  - 表示收到对方的初始序列号
- 第一个ACK通常和第二个SYN合在一个数据包中
  - 三次握手，Three-Way handshake



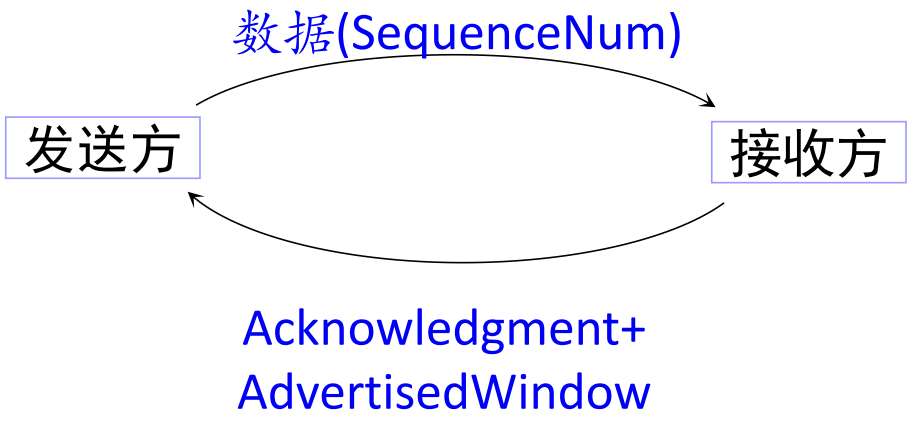
TCP连接断开

- 连接任何一方都可以主动断开连接
  - 发送FIN数据包
  - 表示己方不再发送数据
- 另一端可以继续发送数据
  - 对方仍需要对接收数据进行确认
  - TCP是一个全双工传输协议
- 任何一方都可以发送RST包断开连接
  - 一方发送后不应再有数据传输
  - 正常情况下避免使用RST



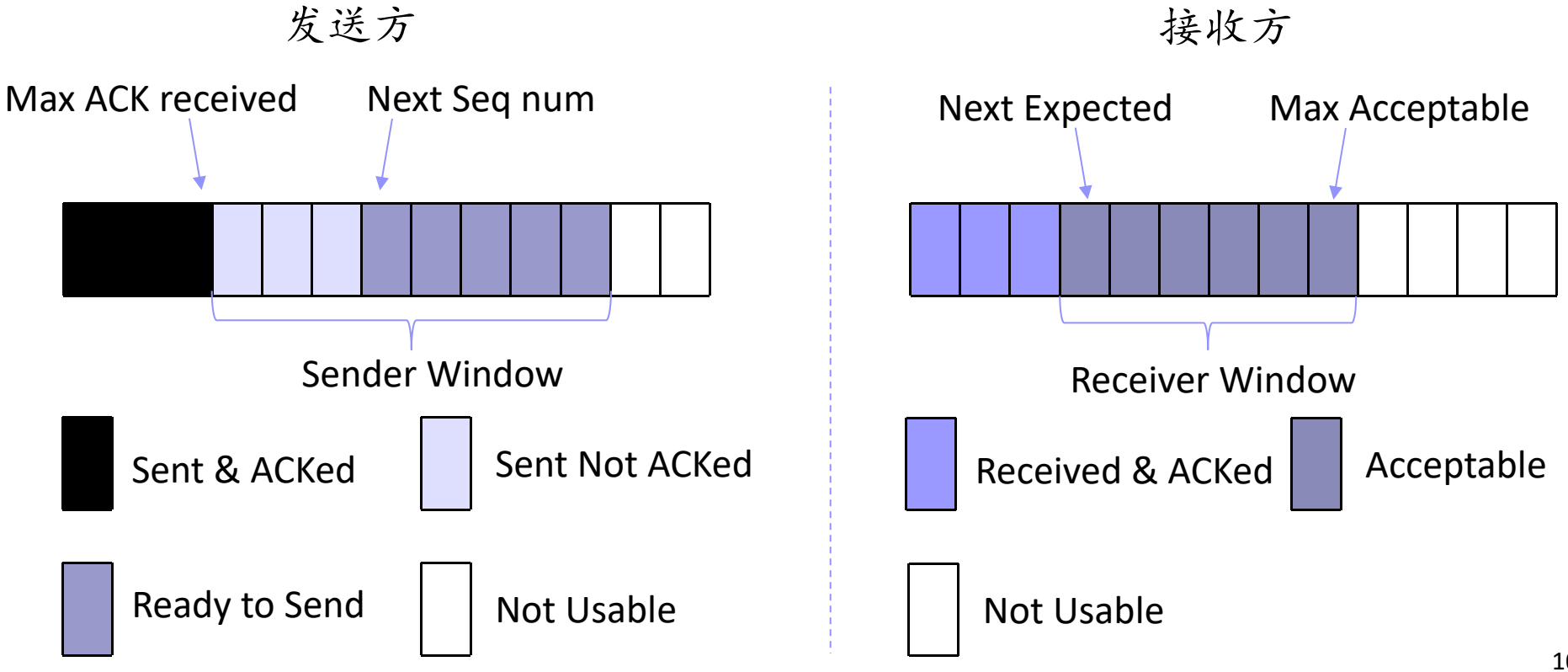
TCP数据传输

- 双方可同时收发数据，以单向为例
  - 有特定序列号值(SequenceNum)的数据，从发送方向接收方流动
    - 每字节顺序编号，每个报文段中的序列号值指的是本报文段所发送的数据的第一个字节的序号
  - 对数据的接收确认(Acknowledgment)、接收窗口大小 (AdvertisedWindow)，由接收方向发送方应答



基于滑动窗口的数据传输

- 基于滑动窗口算法实现可靠的字节流传输
  - 何时发送一个报文段 -- 触发传输
  - 发送速率不能超过接收方接收能力 -- 流量控制



Socket数据结构

- struct tcp_sock是Socket维护TCP连接信息和数据传输控制的核心数据结构
  - 连接双方的IP地址和端口信息
  - 当前的状态(TCP状态)
  - 收发数据序列号
  - 接收数据缓冲区
  - 流控信息
  - 丢包管理 # 下次实验
  - 拥塞控制信息 # 下次实验

IP地址和端口信息

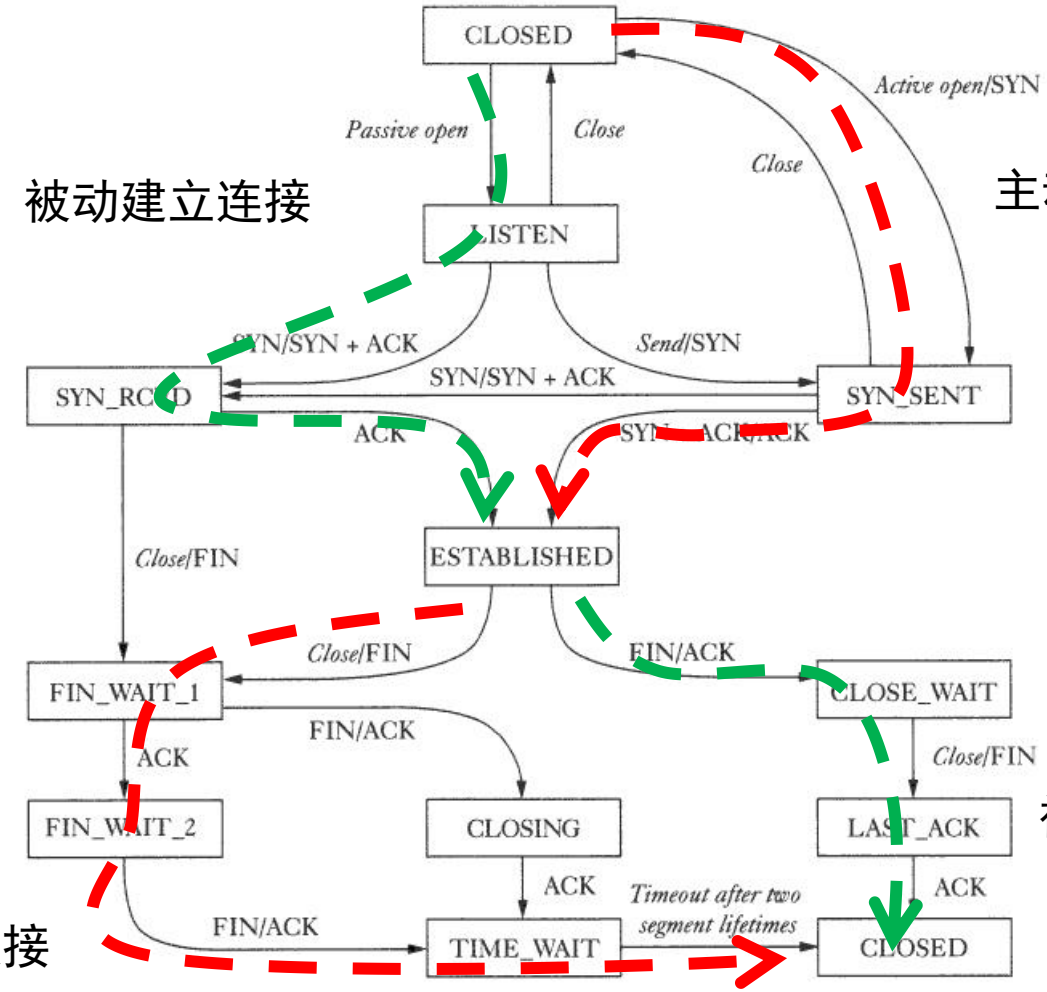
```
struct sock_addr {
    u32 ip;
    u16 port;
} __attribute__((packed));
```

- 本实验只考虑IPv4地址
- 本地存储的地址和端口都是本地字节序
- 本端/对端的地址端口信息构成了四元组，三种情形：
  - 被动建立连接的一方，只有本端地址与端口是确定的
  - 主动建立连接的一方，在建立连接之前，其四元组已经确定
  - 已经建立好的连接，其四元组是确定的

TCP状态

- TCP_CLOSED // 未开始或者已结束的连接
- TCP_LISTEN // 被动建立连接的一方，等待连接请求（被动方）
- TCP_SYN_RECV // 收到对方的SYN数据包（被动方）
- TCP_SYN_SENT // 已发送SYN数据包（主动方）
- TCP_ESTABLISHED // 经过三次握手，双方已经建立连接
- TCP_CLOSE_WAIT // 收到对方的FIN数据包（被动方）
- TCP_LAST_ACK // 发送FIN，等待最后一个ACK（被动方）
- TCP_FIN_WAIT_1 // 发送FIN，主动断开连接（主动方）
- TCP_FIN_WAIT_2 // 收到主动发送FIN对应的ACK（主动方）
- TCP_CLOSING // 发送FIN之后也收到对方的FIN包（主动方）
- TCP_TIME_WAIT // 主动方完成4次挥手操作（主动方）

TCP连接管理和状态迁移



只实现虚线标识的路径过程

TCP收发序列号

- `u32 snd_una;` // 对端连续确认的最大序列号
- `u32 snd_nxt;` // 本端已发送的最大序列号
- `u32 rcv_nxt;` // 本端连续接收的最大序列号
- `u32 iss;` // 本端初始发送序列号

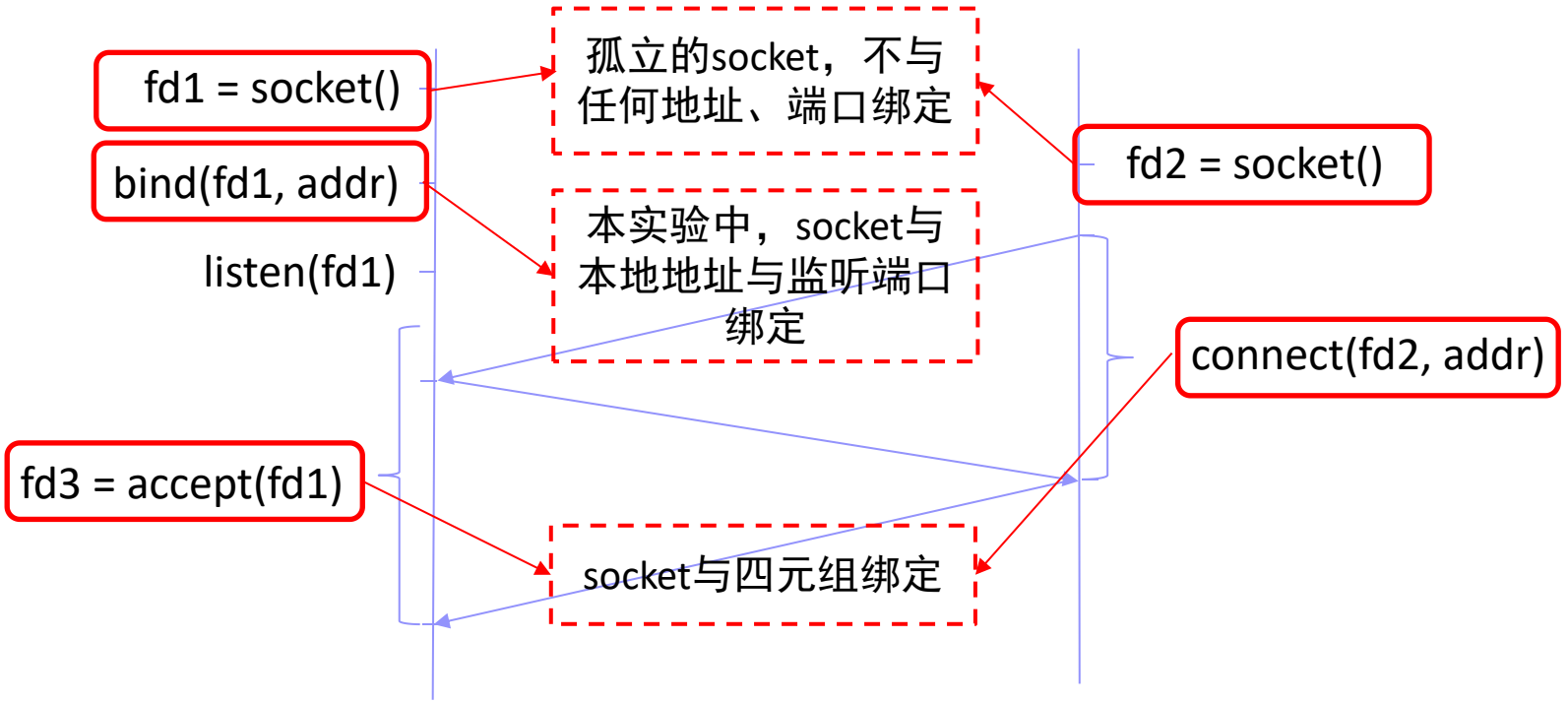
比较序列号大小时，存在整数环绕问题

`snd_una < snd_nxt` ⇒ `(int32_t)(snd_una - snd_nxt) < 0`

在include/tcp.h中有相应宏定义

Socket与元组信息的绑定

- 根据连接所在的不同阶段，Socket绑定不同的元组信息



- 协议栈维护listen_table和established_table两个hash表，来分别组织只绑定源地址、端口的socket和绑定四元组的socket

通过数据包信息查找对应的Socket

```
// the 3 tables in tcp_hash_table
struct tcp_hash_table {
    struct list_head established_table[TCP_HASH_SIZE];
    struct list_head listen_table[TCP_HASH_SIZE];
    struct list_head bind_table[TCP_HASH_SIZE];
};
```

- 对于源目的地址、源目的端口都已经确定下来的socket，按照上述4元组，将hash_list节点hash到established_table
- 对于只知道源地址、源端口的socket，按照上述2元组，将hash_list节点hash到listen_table
- 任何占用一个本地端口的socket，按照该端口号将bind_hash_list 节点hash到bind_table

- 对于一个新到达的数据包，先在established_table中查找相应socket，如果没有找到，再到listen_table中查找相应socket

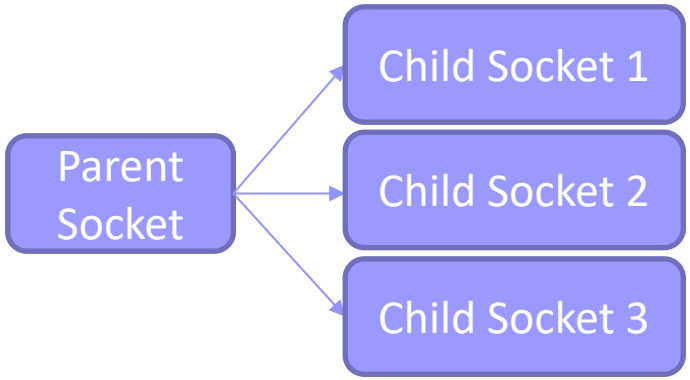
保证所有连接资源最后都能完全释放

- 被动建立连接一方的处理流程

```
struct tcp_sock *tsk, *csk;
tsk = alloc_tcp_sock();
tcp_sock_bind(tsk, &addr);
tcp_sock_listen(tsk, 3);
while ((csk = tcp_sock_accept(tsk)))
    handle_tcp_sock(csk);
```

Parent Socket

Child Socket



Child socket在被tcp_socket_accept返回之前，需要保存在parent socket的队列中：

- listen_queue：未完成三次握手的child socket
- accept_queue：已完成三次握手的child socket

Socket队列

- socket.list
  - 用于将该socket放入到parent socket的队列中
- socket.listen_queue
  - 当被动建立连接的parent socket收到SYN数据包后，会产生一个child socket来服务该连接，放到parent socket的listen_queue队列中
- socket.accept_queue
  - 当接收到三次握手中的最后一个包（ACK）时，在listen_queue中的child socket会放到accept_queue中，等待应用程序读取(tcp_sock_accept)
  - Socket加入到accept_queue中时，parent socket的accept_backlog值加一，离开队列时该值减一，注意accept_backlog < backlog

应用程序与协议栈间的协作

- 应用程序在调用connect, accept函数时，需要等待TCP协议栈完成相应数据包的收发
  - 使用互斥锁+信号机制，模拟阻塞和唤醒操作
    - int sleep_on(struct synch_wait *wait);
    - int wake_up(struct synch_wait *wait);
  - Connect：发送SYN包后，应用程序sleep_on()阻塞，协议栈收到SYN|ACK后调用wake_up()唤醒
  - Accept：accept()时，应用程序sleep_on()阻塞，协议栈收到ACK后调用wake_up()唤醒
- 应用程序调用close断开连接时，不需要阻塞，协议栈会完成后续操作

TCP Sock数据结构

```
struct tcp_sock {
        struct sock_addr local;
        struct sock_addr peer;

        struct tcp_sock *parent;

        int ref_cnt;
        struct list_head hash_list;
        struct list_head bind_hash_list;

        struct list_head listen_queue;
        struct list_head accept_queue;

        int accept_backlog;
        int backlog;
        struct list_head list;
        struct tcp_timer timewait;
        struct tcp_timer retrans_timer;

        struct synch_wait *wait_connect;
        struct synch_wait *wait_accept;
        struct synch_wait *wait_recv;
        struct synch_wait *wait_send;

        struct ring_buffer *rcv_buf;
        struct list_head send_buf;
        struct list_head rcv_ofo_buf;

        int state;
        u32 iss;
        u32 snd_una;
        u32 snd_nxt;
        u32 rcv_nxt;

        u32 recovery_point;
        u32 snd_wnd;
        u16 adv_wnd;
        u16 rcv_wnd;
        u32 cwnd;
        u32 ssthresh;
};
```

TCP Sock相关函数

```
int socket(int, int, int);
struct tcp_sock *alloc_tcp_sock();
int bind(int, struct sockaddr *, socklen_t);
int tcp_sock_bind(struct tcp_sock *, struct sock_addr *);
int listen(int, int);
int tcp_sock_listen(struct tcp_sock *, int);
int connect(int, struct sockaddr *, socklen_t);
int tcp_sock_connect(struct tcp_sock *, struct sock_addr *);
int accept(int, struct sockaddr *, socklen_t *);
struct tcp_sock *tcp_sock_accept(struct tcp_sock *);
int close(int);
void tcp_sock_close(struct tcp_sock *);
int read(int, char *, int);
int tcp_sock_read(struct tcp_sock *, char *, int);
int write(int, char *, int);
int tcp_sock_write(struct tcp_sock *, char *, int);
```

TCP协议栈实现

- 实现TCP数据包处理
  - 如何建立连接、断开连接、处理异常情况
  - 考虑系统协议栈可能会合并数据包，例如
    - 正常流程：User: FIN; Sys: ACK; Sys: FIN; User: ACK
    - 数据包合并：User: FIN; Sys: FIN|ACK; User: ACK
- 实现tcp_sock连接管理函数
  - 类似于socket函数，能够绑定和监听端口，建立和断开连接

建立连接

- 被动建立连接
  - 申请占用一个端口号 （bind操作）
  - 监听该端口号 （listen操作）
  - 收到SYN数据包 -> TCP_SYN_RECV （accept操作，应用阻塞，协议栈完成相应处理后唤醒）
  - 回复ACK并发送SYN数据包
  - 收到ACK数据包 -> TCP_ESTABLISHED

- 主动建立连接
  - 发送目的端口的SYN数据包 -> TCP_SYN_SENT （connect操作，应用阻塞，协议栈完成相应处理后唤醒）
  - 收到SYN 数据包（设置TCP_ACK标志位）
  - 回复ACK数据包 -> TCP_ESTABLISHED

断开连接

- 主动断开
  - 发送FIN包，进入TCP_FIN_WAIT_1状态　　（close操作，非阻塞，后续由协议栈完成）
  - 收到FIN对应的ACK包，进入TCP_FIN_WAIT_2状态
  - 收到对方发送的FIN包，回复ACK，进入TCP_TIME_WAIT状态
  - 等待2*MSL时间，进入TCP_CLOSED状态，连接结束
    - 需要实现定时器线程，定期扫描，适时结束TCP_TIME_WAIT状态的流
- 被动断开
  - 收到FIN包，回复相应ACK，进入TCP_CLOSE_WAIT状态，还可以发送数据
  - 自己没有待发送数据，断开连接，发送FIN包，进入TCP_LAST_ACK状态（close操作，非阻塞，后续由协议栈完成）
  - 收到FIN包对应的ACK，进入TCP_CLOSED状态，连接结束

接收数据包后的处理流程

- 检查TCP校验和是否正确
- 检查是否为RST包，如果是，直接结束连接
- 检查是否为SYN包，如果是，进行建立连接管理
- 检查ack字段，对方是否确认了新的数据
  - 连接管理部分只有SYN和FIN包会确认新数据
- 检查是否为FIN包，如果是，进行断开连接管理

TCP协议栈连接管理主要操作

- 只需要实现TCP Sock的连接管理相关函数

```
• struct tcp_sock *alloc_tcp_sock();
• int tcp_sock_bind(struct tcp_sock *, struct sock_addr *);
• int tcp_sock_listen(struct tcp_sock *, int);
• int tcp_sock_connect(struct tcp_sock *, struct sock_addr *);
• struct tcp_sock *tcp_sock_accept(struct tcp_sock *);
• void tcp_sock_close(struct tcp_sock *);
```

实验内容一：连接管理

- 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_tcp_rst.sh, disable_offloading.sh)，禁止协议栈的相应功能
  - 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_tcp_rst.sh, disable_offloading.sh)，禁止协议栈的相应功能
  - 在h2上运行TCP协议栈的客户端模式，连接至h1，显示建立连接成功后自动断开连接 (./tcp_stack client 10.0.0.1 10001)
- 可以在一端用tcp_stack_conn.py替换tcp_stack执行，测试另一端
- 通过wireshark抓包来来验证建立和断开连接的正确性

流控 (Flow Control)

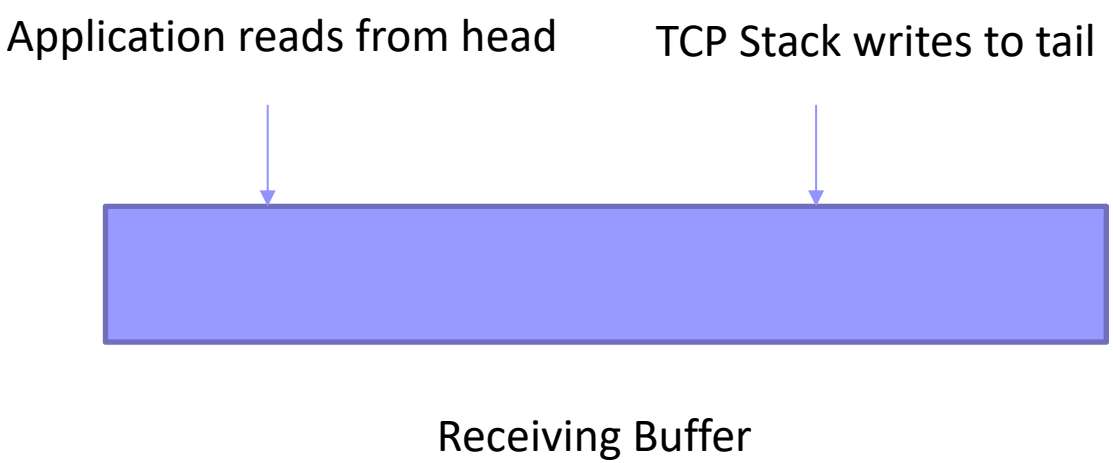
- 为了防止快发送方给慢接收方发数据造成接收崩溃
  - 注意与后续实验中拥塞控制的区别
- 发送方和接收方各自维护一个窗口大小
  - 发送窗口<=接收窗口
  - 发送窗口应该为对端接收窗口和本端拥塞窗口的最小值（本实验只考虑接收窗口）
- 发送方按照发送窗口大小发送数据
- 接收方根据接收窗口大小接收数据
  - 若数据落在窗口以外，直接丢弃
  - 若数据落在窗口以内
    - 收到的是连续数据
      - 将数据交给上层应用，更新窗口边界值
    - 收到不连续的数据
      - 放到buffer中，不更新窗口

数据发送流程

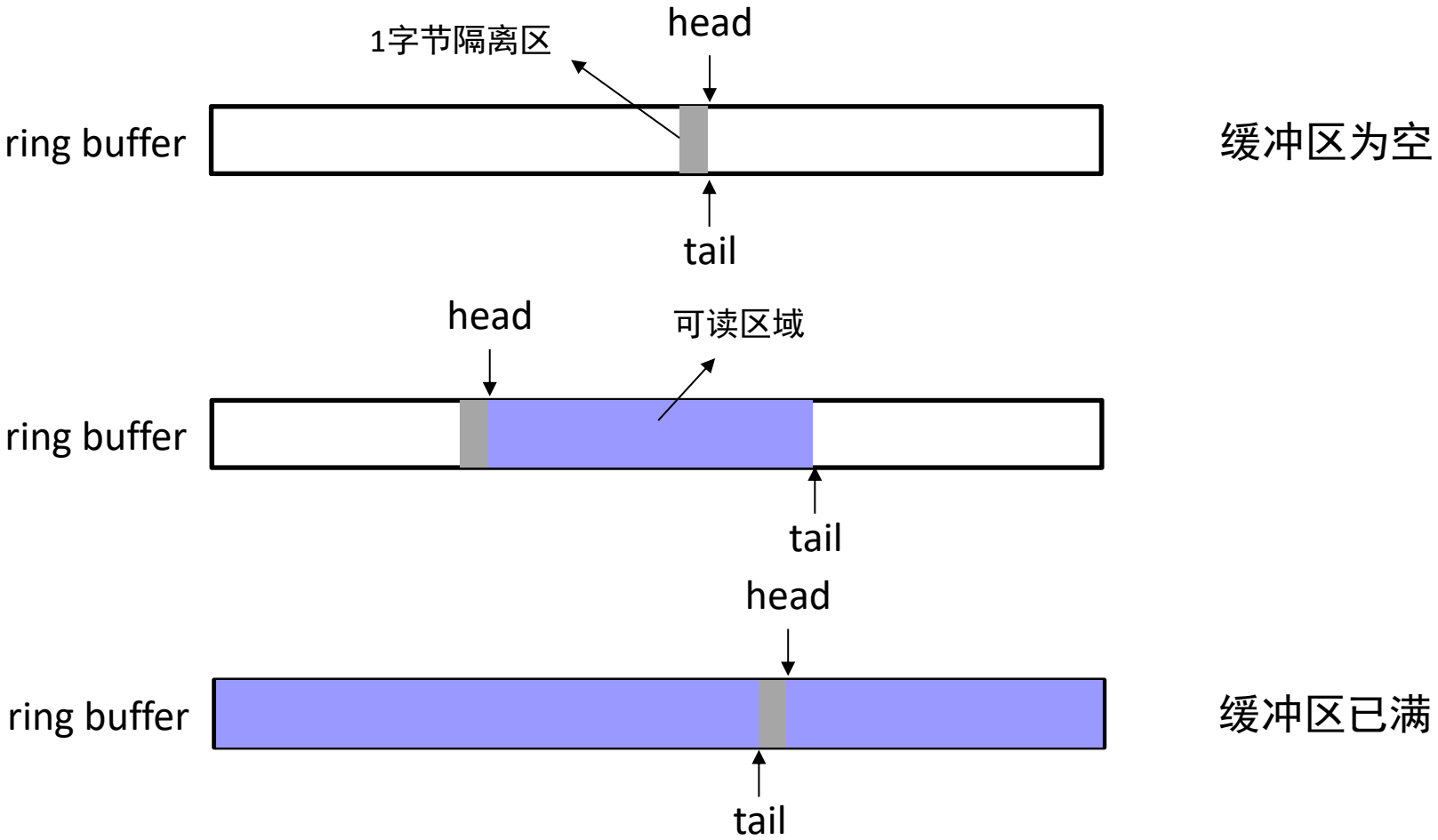
- 发送包含数据的数据包
  - 待发送数据全部存储于上层应用buffer中
  - 如果对端接收窗口（rcv_wnd）允许，则发送数据
  - 每次读取1个数据包大小的数据
    - min(data_len, 1514 - ETHER_HDR_SIZE - IP_HDR_SIZE - TCP_HDR_SIZE)
  - 封装数据包，通过IP层发送函数，将数据包发出去

数据接收和缓存

- TCP协议栈收到数据包后，使用接收缓存来存储相应数据，供应用程序读取
  - 使用环形缓存(ring buffer)来实现
  - 接收缓存大小为本端最大接收窗口大小（64KB）
  - 本端宣告的接收窗口大小（adv_wnd）为接收缓存剩余空间大小
  - 使用锁（pthread_mutex_t）来防止读写冲突，注意不要加到数据结构的尾部



环形缓存示例


1字节隔离区
head
ring buffer
tail
缓冲区为空
head
可读区域
ring buffer
tail
head
ring buffer
tail
缓冲区已满

TCP协议栈数据收发主要操作

- 只需实现tcp_sock相关函数即可，不需要封装socket接口

- int tcp_sock_read(struct tcp_sock *tsk, char *buf, int len);
  - 返回值：0表示读到流结尾，对方断开连接
    -1表示出现错误
    正值表示读取的数据长度

- int tcp_sock_write(struct tcp_sock *tsk, char *buf, int len);
  - 返回值：-1表示出现错误
    正值表示写入的数据长度

TCP数据收发实现

- 实现数据传输
  - 如何将数据封装到数据包并发送
  - 收到数据和ACK时的相应处理
- 实现流量控制
  - 接收方调整本端接收窗口大小（adv_wnd）来表达自己的接收能力
  - 发送方根据对端接收窗口大小（rcv_wnd）来控制自己的发送速率
- 实现tcp_sock相关函数
  - 类似socket函数，能够收发数据

实验内容二：短消息收发

- 参照tcp_stack_trans.py，修改tcp_apps.c，使之能够收发短消息
- 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_trans.py替换其中任意一端，对端都能正确收发数据

实验内容三：大文件传送

- 修改tcp_apps.c(以及tcp_stack_trans.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_trans.py替换其中任意一端，对端都能正确收发数据

附件文件列表

- create_randfile　　# 随机生成待传输文件
- libipstack(32).a
- ip.c
- tcp_apps.c　　# 基于tcp-stack的服务器和客户端程序
- tcp.c　　# TCP协议相关处理函数
- tcp_in.c　　# TCP接收相关函数
- tcp_out.c　　# TCP发送相关函数
- tcp_sock.c　　# tcp_sock操作相关函数
- tcp_stack_conn.py　　# python应用实现，用于测试连接管理
- tcp_stack_trans.py　　# python应用实现，用于测试短消息收发
- tcp_timer.c　　# TCP定时器
- tcp_topo.py　　# Mininet拓扑，包含两个节点，无丢包环境