网络传输机制实验

冯吕

University of Chinese Academy of Sciences

2018 年 7 月 13 日

主要内容

1. 传输实验一：连接管理与 socket 实现
2. 传输实验二：数据收发
3. 传输实验三：可靠传输

建立连接

TCP 传输机制中连接建立的关键在于三次握手.

- 主动建立连接：
  - 发送目的端口的 *SYN* 数据包 → *TCP_SYN_SENT* (*connect*);
  - 收到 *SYN* | *ACK* 数据包: 第二次握手;
  - 回复 *ACK* 数据包 → *TCP_ESTABLISHED*;
- 被动建立连接：
  - 申请占用一个端口号： *bind*;
  - 监听端口号： *listen*;
  - 收到 *SYN* 数据包 → *TCP_SYN_RECV* : 第一次握手;
  - 回复 *SYN* | *ACK* 数据包;
  - 收到 *ACK* 数据包 → *TCP_ESTABLISHED* : 第三次握手;

主动建立连接

主动连接远程 *tcp socket*:

```
int tcp_sock_connect(struct tcp_sock *tsk, struct sock_addr *skaddr)
{
    tsk->peer.ip = ntohl(skaddr->ip);
    tsk->peer.port = ntohs(skaddr->port);
    tsk->local.ip = ((iface_info_t *)(instance->iface_list.next))->ip;
    if(tcp_sock_set_sport(tsk, tcp_get_port()) < 0){
        printf("Set port failed.\n");
        return -1;
    }

    tsk->iss = tcp_new_iss();
    tsk->snd_nxt = tsk->iss;
    tcp_set_state(tsk, TCP_SYN_SENT);
    tcp_hash(tsk);
    tcp_send_control_packet(tsk, TCP_SYN);
    sleep_on(tsk->wait_connect);

    return 1;
}
```

Listen

tcp_sock_listen

设置 *backlog*, 进入 *TCP_LISTEN* 状态，同时将 *tcp sock hash* 到 *listen table* 上：

```
int tcp_sock_listen(struct tcp_sock *tsk, int backlog)
{
    tsk->backlog = backlog;
    tcp_set_state(tsk, TCP_LISTEN);
    tcp_hash(tsk);
    return 0;
}
```

被动建立连接

tcp_sock_accept

如果 *accept queue* 不为空，弹出第一个 *tcp sock*，否则，*sleep on wait_accept*:

```
struct tcp_sock *tcp_sock_accept(struct tcp_sock *tsk)
{
    if (!tsk->accept_backlog){
        sleep_on(tsk->wait_accept);
    }
    if (tsk->accept_backlog){
        struct tcp_sock *accept_tsk = tcp_sock_accept_dequeue(tsk);
        tcp_set_state(accept_tsk, TCP_ESTABLISHED);
        return accept_tsk;
    }
    return NULL;
}
```

Look Up

tcp_sock_lookup_established

对于一个新到达的数据包，需要先在 *established table* 中查找 *socket*:

```
struct tcp_sock *tcp_sock_lookup_established(u32 saddr, u32 daddr,
      u16 sport, u16 dport)
{

    int key = tcp_hash_function(saddr, daddr, sport, dport);
    struct tcp_sock *sk_pos, *sk_q;
    list_for_each_entry_safe(sk_pos, sk_q,
          &tcp_established_sock_table[key], hash_list){
        if (saddr == sk_pos->local.ip && daddr == sk_pos->peer.ip
            && sport == sk_pos->local.port
            && dport == sk_pos->peer.port){
            return sk_pos;
        }
    }
    return NULL;
}
```

Look Up

tcp_sock_lookup_listen

如果在 *established table* 中没有找到对应的 *socket*，再到 *listen table* 中查找：

```
struct tcp_sock *tcp_sock_lookup_listen(u32 saddr, u16 sport)
{
    u8 key = tcp_hash_function(0, 0, sport, 0);
    struct tcp_sock *sk_pos, *sk_q;
    list_for_each_entry_safe(sk_pos, sk_q, &tcp_listen_sock_table[key],
            hash_list){
        if (sk_pos->local.port == sport){
            return sk_pos;
        }
    }
    return NULL;
}
```

tcp_state_listen

被动连接的一方处于 *TCP_LISTEN* 状态时收到包 (第一次握手)：

```
void tcp_state_listen(struct tcp_sock *tsk,
      struct tcp_cb *cb, char *packet)
{
    if (cb->flags & TCP_SYN){
        struct tcp_sock *child_sock = alloc_tcp_sock();
        child_sock->local.ip = cb->daddr;
        child_sock->local.port = cb->dport;
        child_sock->peer.ip = cb->saddr;
        child_sock->peer.port = cb->sport;
        child_sock->parent = tsk;
        child_sock->rcv_nxt = cb->seq_end;
        child_sock->iss = tcp_new_iss();
        child_sock->snd_nxt = child_sock->iss;
        struct sock_addr skaddr = {htonl(child_sock->local.ip), htons(chi
ld_sock->local.port)};
        tcp_sock_bind(child_sock, &skaddr);
        tcp_set_state(child_sock, TCP_SYN_RECV);
        list_add_tail(&child_sock->list,
              &tsk->listen_queue);
        tcp_send_control_packet(child_sock,
              TCP_SYN | TCP_ACK);
        tcp_hash(child_sock);
    }
    else {
        tcp_send_reset(cb);
    }
}
```

tcp_state_syn_sent

主动连接的一方处于 *TCP_SYN_SENT* 状态时收到包 (第二次握手):

```
void tcp_state_syn_sent(struct tcp_sock *tsk, struct tcp_cb *cb,
     char *packet)
{
    if (cb->flags & (TCP_SYN | TCP_ACK)){
        tsk->rcv_nxt = cb->seq_end;
        tsk->snd_una = cb->ack;
        tcp_send_control_packet(tsk, TCP_ACK);
        tcp_set_state(tsk, TCP_ESTABLISHED);
        wake_up(tsk->wait_connect);
    }
    else {
        tcp_send_reset(cb);
    }
}
```

tcp_state_syn_recv

被动连接的一方处于 *TCP_SYN_RECV* 状态时收到包 (第三次握手):

```
void tcp_state_syn_recv(struct tcp_sock *tsk, struct tcp_cb *cb,
      char *packet)
{
    if (cb->flags & TCP_ACK){
        tcp_sock_accept_enqueue(tsk);
        wake_up(tsk->parent->wait_accept);
    }
    else {
        tcp_send_reset(cb);
    }
}
```

关闭连接

- 主动关闭：
  - 发送 *FIN* 包，进入 *TCP_FIN_WAIT_1* 状态；
  - 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；
- 被动关闭：
  - 收到 *FIN* 包，回复相应的 *ACK*，进入 *TCP_CLOSE_WAIT* 状态；
  - 当自己没有待发数据时，发送 *FIN* 包，进入 *TCP_LAST_ACK* 状态；
  - 收到 *FIN* 包对应的 *ACK*，进入 *TCP_CLOSED* 状态，连接结束；

接收数据包处理流程

tcp_process：需要根据 *socket* 当前所处的状态和包的类型进行对应的处理：

- 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；
- 检查是否为 FIN 包，如果是，进行断开连接管理；

无丢包情况下的数据收发

本次实验需要实现无丢包环境下的数据收发：

- *tcp_sock_write* 封装数据包，然后将数据包从 *IP* 层发送出去，*socket* 收到数据包后将数据写到接收缓存中，同时回复 *ACK*;
- *tcp_sock_read* 从接收缓存中读取数据到上层应用 *buff* 中;
- 读写过程中需要使用锁来对缓存进行互斥访问：在 *ring_buffer* 中添加一个锁;

tcp_sock_write

将上层应用 *buff* 中的数据封装成数据包发送出去，如果当前发送窗口为 0，则 *sleep on wait_send*:

```
int tcp_sock_write(struct tcp_sock *tsk, char *buf, int len){

    int sent_len = 0;
    while(sent_len < len){
        if (tsk->snd_wnd == 0){
            sleep_on(tsk->wait_send);
        }
        int data_len = min(tsk->snd_wnd, min(len, 1500
            - ETHER_HDR_SIZE - IP_BASE_HDR_SIZE - TCP_BASE_HDR_SIZE));
        int pkt_size = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE +
            TCP_BASE_HDR_SIZE + data_len;

        char *packet = (char *) malloc (pkt_size);
        memset(packet, 0, pkt_size);
        memcpy(packet + ETHER_HDR_SIZE+IP_BASE_HDR_SIZE+
                    TCP_BASE_HDR_SIZE, buf + sent_len, data_len);

        tcp_send_packet(tsk, packet, pkt_size);
        sent_len += data_len;
    }
    return 1;
}
```

tcp_sock_read

从 *socket* 的接收缓存中读取数据，如果当前没有数据，则 *sleep on wait_recv*:

```
int tcp_sock_read(struct tcp_sock *tsk, char *buf, int len){
    if (ring_buffer_empty(tsk->rcv_buf)){
        sleep_on(tsk->wait_recv);
    }
    pthread_mutex_lock(&tsk->rcv_buf->rw_lock);
    int read_len = read_ring_buffer(tsk->rcv_buf, buf, len);
    pthread_mutex_unlock(&tsk->rcv_buf->rw_lock);
    return read_len;
}
```

收到数据包的处理

- 当收到包含数据的数据包时，将数据写入到 *ring_buffer* 中，同时 *wake_up wait_recv*, 然后回复 *ACK*;
- 当收到 *ACK* 数据包时，*update sending window*;

可靠传输

- 实验三需要实现在丢包环境下的可靠传输。
- 在 *tcp_sock* 中增加三个成员：

```
1  struct tcp_timer retrans_timer; // 超时重传
2  //定时器
3  struct list_head send_buf;//未确认数据
4  struct list_head rcv_ofo_buf;//不连续数据
5  //存于send_buf/ofo_buf中的数据结构
6  struct send_packet{
7      struct list_head list;
8      char *packet;
9      int len;
10 };
11 struct ofo_packet{
12     struct list_head list;
13     char *packet;
14     int len;
15     int seq num;
```

修改 *send packet*

- 修改 *tcp_send_packet*：当发送一个 *packet* 时，将其存到 *send_buf* 中，同时启动一个重传定时器;

```
1 struct send_packet *send_pkt = (struct
2 send_packet *)malloc(sizeof(struct send_packet));
3 send_pkt->packet = (char *)malloc(len);
4 send_pkt->len = len;
5 memcpy(send_pkt->packet, packet, len);
6 list_add_tail(&send_pkt->list, &tsk->send_buf);
7 tcp_set_retrans_timer(tsk);
```

- 修改 *tcp_send_control_packet*：如果发送的是 *SYN* | *FIN* 包，也需要将其存到 *send_buf* 中，直到收到 *ACK*;

tcp_timer

- 在 *tcp_timer* 中增加一项：重传次数;
- 设置重传定时器和关闭定时器：

```
void tcp_set_retrans_timer(struct tcp_sock *tsk){
    struct tcp_timer *timer = &tsk->retrans_timer;

    timer->type = 1;
    timer->timeout = TCP_RETRANS_INTERVAL;
    timer->retrans_number = 0;

    list_add_tail(&timer->list, &timer_list);
}

void tcp_remove_retrans_timer(struct tcp_sock *tsk){
    list_delete_entry(&tsk->retrans_timer.list);
}
```

修改 *tcp_scan_timer_list*

- (1) 判断定时器类型，如果类型为 *wait*，则根据是否 *timeout* 关闭即可，如果类型为 *retrans*，转 (2);
- (2) 判断重传次数是否小于 3，如果不小于，则关闭该 *timer* 对应的 *socket*，否则转 (3);
- (3) 重传 *send_buf* 中的包，更新定时器：
  $timeout * 2, retrans\_number + 1$

收到 *ACK*

当收到 *ACK* 时，将 *send_buf* 中 *sep_end* < *ack* 的包移除，同时更新定时器：

```
void remove_ack_data(struct tcp_sock *tsk, int ack_num){
    tcp_remove_retrans_timer(tsk);
    struct send_packet *pos, *q;
    list_for_each_entry_safe(pos, q, &tsk->send_buf, list){
        struct tcphdr *tcp = packet_to_tcp_hdr(pos->packet);
        struct iphdr *ip = packet_to_ip_hdr(pos->packet);
        if (ack_num >= ntohl(tcp->seq)){
            tsk->snd_wnd += (ntohs(ip->tot_len) -
                        IP_HDR_SIZE(ip) - TCP_HDR_SIZE(tcp));
            free(pos->packet);
            list_delete_entry(&pos->list);
        }
    }
    if (!list_empty(&tsk->send_buf)){
        tcp_set_retrans_timer(tsk);
    }
}
```

收到连续数据

如果收到连续的数据：$cb->seq == tsk->rcv\_nxt$

- (1) 将数据写到 *ring_buffer* 中：*write_ring_buffer*()
- (2)$tsk->rcv\_nxt = cb->seq\_end$，并 *wake_up wait_recv*
- (3) 判断 *ofo buffer* 中是否出现连续数据 ($packet->seq == tsk->rcv\_nxt$)，则将数据从 *ofo buffer* 中写到 *ring buffer* 中。

收到不连续数据

如果收到不连续数据：$cb->seq > tsk->rcv\_nxt$，则将数据包存到 *ofo_buffer* 中：

```
struct ofo_packet *buf_pac = (struct ofo_packet *)
        malloc(sizeof(struct ofo_packet));
buf_pac->packet = (char *)malloc(cb->pl_len);
buf_pac->len = cb->pl_len;
buf_pac->seq_num = cb->seq;
memcpy(buf_pac->packet, cb->payload, cb->pl_len);
list_add_tail(&buf_pac->list, tsk->rcv_ofo_buf);
```

谢谢！