天津大学

TrivialTCP 设计文档

学　　院：　智能与计算学部
专　　业：　计算机科学与技术
姓　　名：　齐呈祥 高树韬

2021年 8月 25日

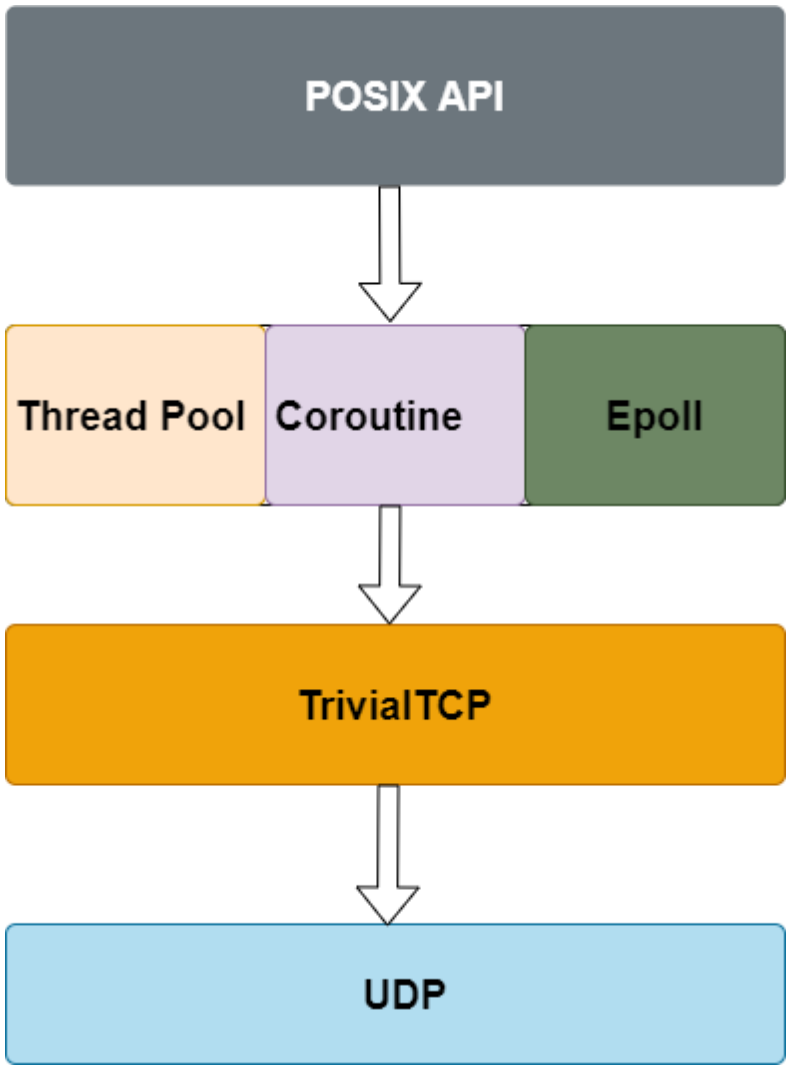
TrivialTCP

TrivialTCP 是一个开源的使用C语言编写的TCP协议开源库。

我们使用 UDP 来模拟 IP 层去收发 packet，其中 TrivialTCP 是本项目中最核心的部分，它保证了能够进行可靠传输。

其中，我们也打算去实现一些高性能的组件运行在 TrivialTCP 上层，例如线程池、协程以及 Epoll 机制，目前线程池和协程已经实现，Epoll 正在实现中。

同时，我们在顶层模块暴露了类似 POSIX 标准的 API 供上层应用使用，因此我们可以为上层的应用程序提供服务。

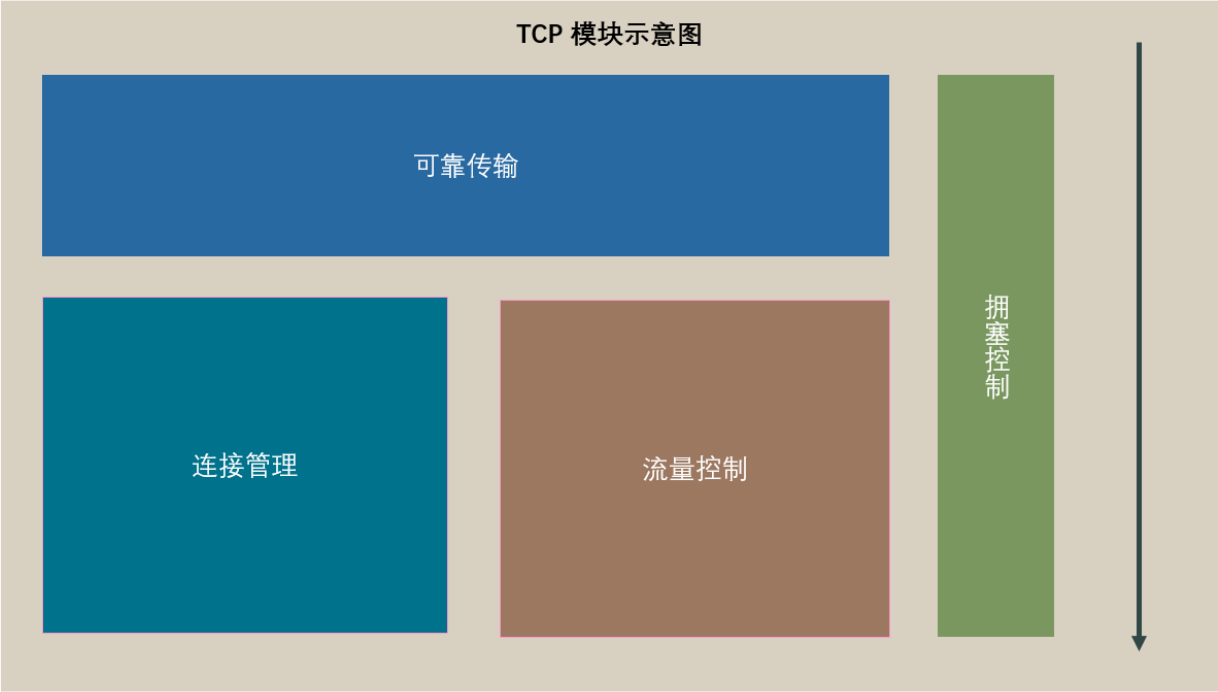


实现模块

- [x] 环境搭配
- [x] 线程池
- [x] 协程
- [ ] Epoll机制
- [x] 连接管理
- [x] 计时器

- [x] 系统API
- [x] 可靠传输
- [x] 流量控制
- [x] 拥塞控制

TCP模块预览



模块简介

1 可靠传输

通过ACK SEQ来确保TCP段的正确接收，通过校验和等方式确认收到的段是否正确，通过五元组确认是否收到了正确连接的TCP段。

2 流量控制

通过改善窗口算法，延迟ACK算法以及Nagle算法，根据接收方的窗口大小，对TCP段的发送进行管理，防止缓冲区收到过多的数据而被迫丢弃或者SWS现象（愚蠢窗口综合征）。

3 连接管理

TCP协议要求客户端和服务端通过三次握手发起连接。首先，由客户端调用 connect() 方法主动向服务端发起连接，向服务端发送带有 SYN 标志段的 packet，当服务端接受到分组后向客户端返回带有 ACK 标志位，随后客户端向服务端发送带有 ACK 标志位的分组，至此三次握手正式完成。

断开连接与连接过程类似，同样由一端发起断开连接请求（即发送 FIN 标志位的报文），双方经过协商与状态机转换后最终同时关闭连接并释放资源。

4 拥塞控制

TCP的四种拥塞控制算法

- 慢启动
- 拥塞控制
- 快重传
- 快恢复

模块详细设计

1 可靠传输

1.1 完全可信信道上的可靠传输

如果信道完全可靠，那么可靠传输就不成问题了，此时的可靠传输非常简单。发送方只需要将数据放到信道上它就可以可靠的到达接收方，并由接收方接收。但是这种信道是完全理想化的，不存在的。

1.2 会出现比特错误的信道上的可靠传输

更现实一点的信道是会发生比特错误的，假设现在需要在除了会出现比特错误之外，其它的特性和完全可靠信道一样的信道上进行可靠传输.

为了实现该信道上的可靠传输：

- 接收方：需要确认信息是否就是发送方所发送的，并且需要反馈是否有错误给发送方
- 发送方：需要在发送的信息中添加额外信息以使得接收方可以对接收到的信息是否有错误进行判断，并且需要接收接收方的反馈，如果有错误发生就要进行重传

因而在这种信道上进行传输需要三种功能：

- 差错检测：发送方提供额外信息供接收方进行校验，接收方进行校验以判断是否有错误发生。网络协议一般采用校验和来完成该任务
- 接收方反馈：接收方需要将是否有错误发生的信息反馈给发送方，这就是网络协议中最常见的ACK（确认）/NAK（否定的确认）机制
- 发送方重传：在出现错误时，发送方需要重传出错的分组。重传也是网络协议中极常见的机制。

上述机制还有问题，它没有考虑接收方的反馈出现比特错误 即**反馈受损**的情形。采用上述机制，在反馈受损时，发送方可以了解到这个反馈信息出现了错误，但是它无法知道反馈的是什么样的信息，因此也就无法知道自己该怎么应对。

这可以有两种解决办法：

- 发送方提供足够多的信息，使得接收方不仅可以检测比特错误，而且可以恢复比特错误。这在仅会发生比特错误的信道上是理论可行的，代价是需要大量额外的信息。
- 如果收到了受损的反馈，则都认为是出现了错误，就进行重传。但是这时就可能引入冗余的分组，因为被重传的分组可能已经正确的被接收了。

网络协议中广泛采用的是第2种解决方案，冗余分组可以通过一种简单的机制来解决，这就是分组序列号。被发送的每个分组都有一个序列号，接收方只需要检测该序列号就可以知道分组是否是冗余的。

在引入序列号后，该机制已经可以在这种信道上工作了。不过它还可以做一点变化，有些网络协议中并不会产生否定的确认（即报告发送方出现了错误），它采用的是继续为 已经为之发送过ACK的最后一个正确接收的分组 发送ACK。当发送方收到冗余的ACK时就知道跟在被冗余ACK确认的分组之后的分组没有被正确接收，这就达到了NAK所要的效果。

在我们的实现中，我们在每个 `socket` 中维护了两个窗口：发送窗口和接收窗口，其中发送窗口维护 `base` 和 `nextseq` 两个 field，分别用来表示当前发送但仍未收到 ACK 的第一个 pakcet 的序列号以及下一个将要发送 packet 的序列号。借此可以实现 GBN 窗口移动算法。接收窗口维护 `expectd_seq` 用来判断发送方发来的分组是否失序，只要当发送方发来的分组的序列号等于 `expected_seq` 的时候才接收。

我们进行通信的流程如下所示：

```
用户发送数据，TrivialTCP 将数据放入发送缓冲区中 ---> 发送线程不断轮询发送缓冲区，当发现
缓冲区中存在数据，将其封装头部，送入下层协议，开启定时器 --->
接收方收到分组，判断检验和是否正确，若不正确则丢弃 ---> 判断 seq 的值是否与
expected_seq 相同，若不相同则丢弃 packet -->
关闭定时器，并送入接收缓冲区中
```

其中，倘若计时器发现超时，则会调用回调函数进行处理。

涉及到的方法：

- `int tcp_check(tju_packet_t* pkt)`：判断检验和是否正确，若正确返回 TRUE，否则返回 FALSE。
- `int tcp_check_seq(tju_packet_t* pkt, tju_tcp_t* sock)`：检查序列号是否正确。
- `static unsigned short tcp_compute_checksum(tju_packet_t* pkt)`：计算检验和。
- `void* tcp_send_stream(void* arg)`：使用 GBN 算法，轮询发送缓冲区，并将缓冲区内容发送给下层协议。

2 连接管理

2.1 三次握手

TCP协议要求客户端和服务端通过三次握手发起连接。首先，由客户端调用 `connect()` 方法主动向服务端发起连接，向服务端发送带有 `SYN` 标志段的 packet，当服务端接受到分组后向客户端返回带有 `ACK` 标志位，随后客户端向服务端发送带有 `ACK` 标志位的分组，至此三次握手正式完成。

当服务端接受到 `SYN` packet 之后，会将新建客户端 socket并将其加入到半连接队列中 `syns_socks`中，我们 `socket` 队列的定义如下:

```
typedef struct sock_node {
	tju_tcp_t* data;
	struct sock_node* next;
} sock_node;

typedef struct sock_queue {
	int size;
	sock_node* base;
} sock_queue;
```

其中我们为 `sock_queue` 实现了队列的各种基础算法。

当服务端再次接受到客户端发来的 `ACK` 报文时，则将半连接队列中的 `scoket` 发送给全连接队列 `accept_socks`，此时 `accept()` 方法中检测到 `accept_socks` 中收到的分组，随后则将监听 `socket` 状态修改为 `ESTABLISHED`,并加入 `established_socks` 中。

客户端的处理过程与服务端类似， `connect()` 主动向服务端发送报文，随后阻塞等待直到 `connect_sock` 不为 NULL。

其中关于接收报文的过程则在一个子线程中进行。当内核检测到接受到分组后，则将其交给 `onTCPPocket()` 来处理分组，其中， `onPocket()` 分为不同情况来处理分组:

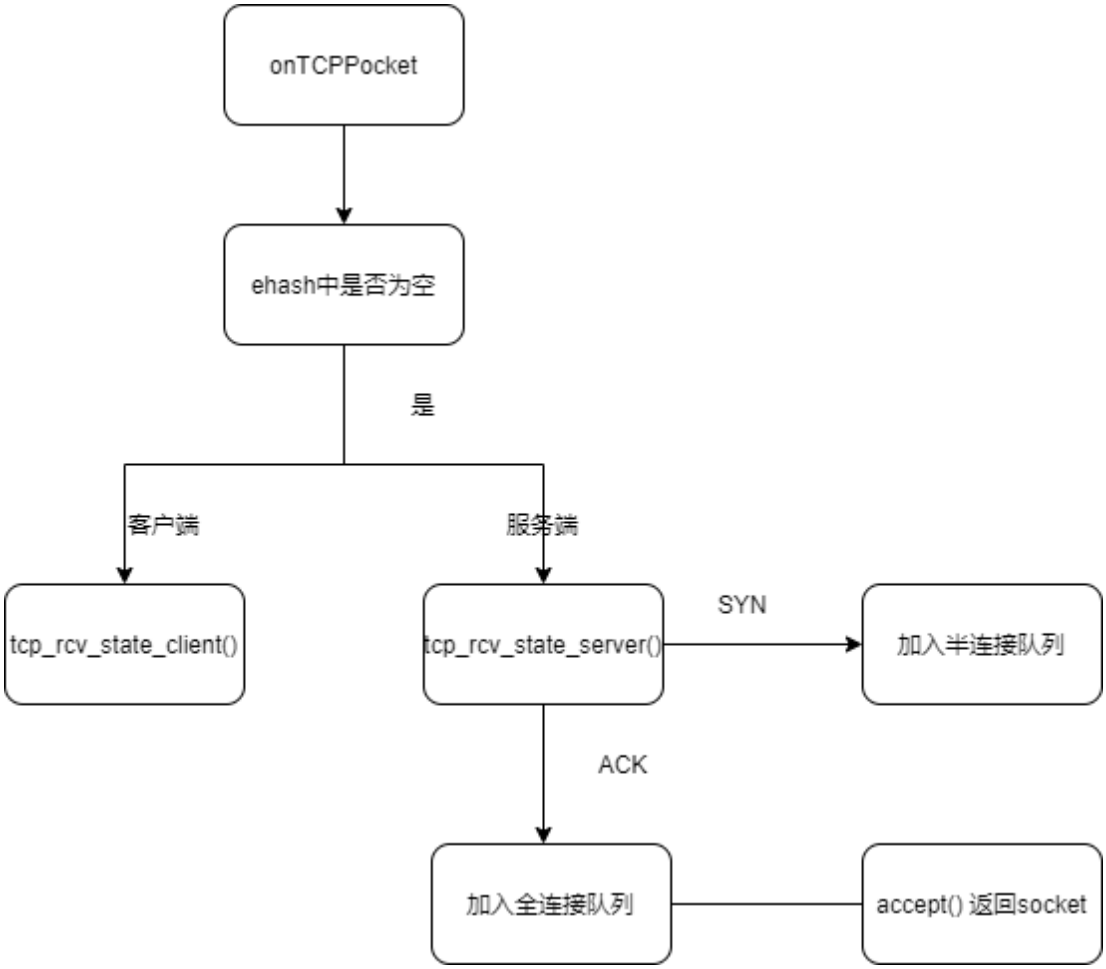
```
/*
模拟Linux内核收到一份TCP报文的处理函数
*/
int onTCPPocket(char* pkt){
	// 当我们收到TCP包时 包中 源IP 源端口 是发送方的 也就是我们眼里的 远程(remote) IP 和端口
	uint16_t remote_port = get_src(pkt);
	uint16_t local_port = get_dst(pkt);
	printf("get a pkt rwnd is %d \n", get_advertised_window(pkt));
	// remote ip 和 local ip 是读IP 数据包得到的 仿真的话这里直接根据hostname判断
	// 获取是server还是client
	int is_server;
	char hostname[8];
	gethostname(hostname, 8);
	uint32_t remote_ip, local_ip;
	if(strcmp(hostname,"server")==0){ // 自己是服务端 远端就是客户端
		local_ip = inet_network("10.0.0.3");
		remote_ip = inet_network("10.0.0.2");
		is_server = 1;
	}else if(strcmp(hostname,"client")==0){ // 自己是客户端 远端就是服务端
		local_ip = inet_network("10.0.0.2");
		remote_ip = inet_network("10.0.0.3");
		is_server = 0;
	}

	tju_packet_t* packet = buf_to_packet(pkt);
```

```
    if(!tcp_check(packet)) {
        printf("tcp check error.\n");
        return -1;
    }
    if(packet->data != NULL) {
        free(packet->data);
    }
    free(packet);

    int hashval;

    // 首先查找已经建立连接的socket哈希表
    // 根据4个ip port 组成四元组 查找有没有已经建立连接的socket
    hashval = cal_hash(local_ip, local_port, remote_ip, remote_port);
    if (established_socks[hashval] != NULL) {
        // 这里应当判断是否发送FIN packet, 或者socket的状态不是ESTABLIED
        int new_hash = cal_hash(local_ip, local_port, 0, 0);
        if(is_server && (is_fin(pkt) || listen_socks[new_hash]->state !=
ESTABLISHED)) {
            return tcp_state_close(listen_socks[new_hash], pkt);
        }else if(!is_server &&(is_fin(pkt) || connect_sock->state != ESTABLISHED)) {
            return tcp_state_close(connect_sock, pkt);
        }else {
            return tju_handle_packet(established_socks[hashval], pkt);
        }
    }

    tju_sock_addr conn_addr;
    conn_addr.ip = remote_ip;
    conn_addr.port = remote_port;


    hashval = cal_hash(local_ip, local_port, 0, 0);
    // 没有的话再查找监听中的socket哈希表
    if (listen_socks[hashval] != NULL && is_server) {
        // 监听的socket只有本地监听ip和端口 没有远端
        return tcp_rcv_state_server(listen_socks[hashval], pkt, &conn_addr);
    }

    if (connect_sock != NULL && !is_server) {
        return tcp_rcv_state_client(connect_sock, pkt, &conn_addr);
    }

    // 都没找到 丢掉数据包
    printf("找不到能够处理该TCP数据包的socket, 丢弃该数据包\n");
    return -1;
}
```

其中，`tcp_rcv_state_server()` 和 `tcp_rcv_state_client()` 方法来分别处理服务端与客户端握手时的状态机转换过程。

涉及到的方法：

- `int tcp_rcv_state_server(tju_tcp_t* sock, char* pkt, tju_sock_addr* conn_addr);`：服务端三次握手的状态转换过程。 参数：服务端socket， 收到的分组，待连接的服务端socket。返回值为处理是否成功，成功返回0，失败返回1。
- `int tcp_rcv_state_client(tju_tcp_t* sock, char* pkt, tju_sock_addr* conn_sock)`：客户端三次握手的状态转换过程。 参数：客户端socket，收到的分组，待连接的服务端socket。返回值为处理是否成功，成功返回0，失败返回1。



2.2 断开连接（四次挥手）

断开连接与连接过程类似，同样由一端发起断开连接请求（即发送 `FIN` 标志位的报文），双方经过协商与状态机转换后最终同时关闭连接并释放资源。具体实现细节这里不再进行赘述。

涉及到的方法：

- `int tcp_state_close(tju_tcp_t* local_sock, char* recv_pkt)`：关闭连接过程的状态机转换。参数： 本地socket， 接收到的分组。返回值为处理是否成功，成功返回0，失败返回-1。

3 定时器的设计

当应用程序创建 socket 时，我们将会调用 `tcp_init_timer()` 为计时器进行初始化，其中 `timeout` 域先设置为1，注册 `callback`，随后当我们开始传输分组时，我们调用回调函数开始计时。

当我们受到 ACK 之后，我们需要调用 `tcp_ack_update_rtt()` 来更新 RTT，其中 `tcp_ack_update_rtt()` 实现如下：

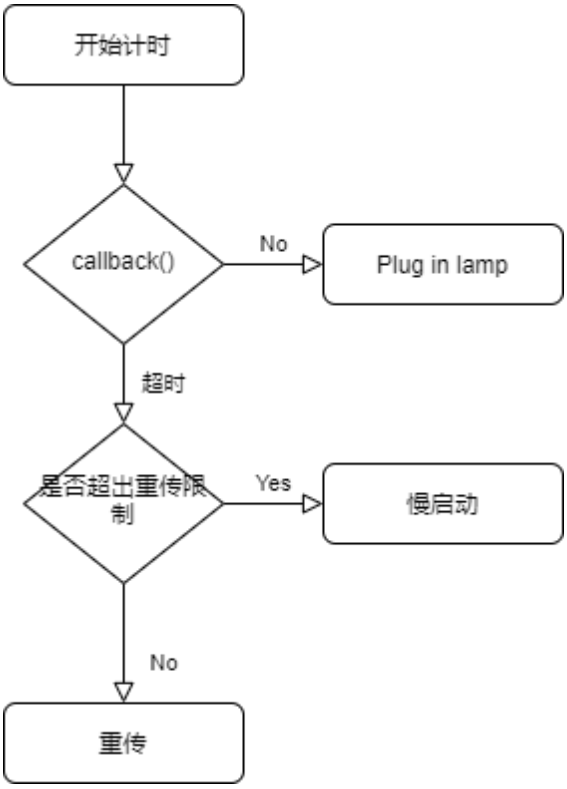
```
int tcp_ack_update_rtt(tju_tcp_t* sock, float seq_rtt_us, float sack_rtt_us) {

      /* Prefer RTT measured from ACK's timing to TS-ECR. This is because
      * broken middle-boxes or peers may corrupt TS-ECR fields. But
      * Karn's algorithm forbids taking RTT if some retransmitted data
      * is acked (RFC6298).
      */
     if (seq_rtt_us < 0)
         seq_rtt_us = sack_rtt_us;

      tcp_set_estimator(sock, seq_rtt_us);
      tcp_set_rto(sock);
      return 0;
}
```

我们需要分别对 `extimator_rtt` 和 `dev_rtt` 进行更新之后再去设置 `timeout`。

定时器的结构设计如下所示：

```
typedef struct rtt_timer_t {
      float estimated_rtt;
      float dev_rtt;
      float timeout;
      void (*callback)(tju_tcp_t* sock);
} rtt_timer_t;
```

其中包含定时器计算 RTT 的基础数据域以及回调函数 `callback()`，其中，我们需要将回调函数注册为 `tcp_write_timer_handler()`，`tcp_write_timer_handler()` 的处理流程为开始计时即调用函数，倘若在规定时间内未收到 ACK，则根据 socket 当前的状态进行处理，例如超时则需要重传，倘若超出了重传次数，则需要启用慢启动重新开始。下图为该过程的一个图示：



关于定时器如何监测是否超时，首先，我们调用 `time_after()` 来判断是否超时，在该函数中我们新建一个线程开始计时，并通过通道监测信号量：

```
// 检查是否超时
void* tcp_check_timeout(void* arg) {
      // 这里我们异步监测是否超时，倘若超时则调用回调函数
      // 否则监测到中断信号返回
      tju_tcp_t* sock = (tju_tcp_t*)arg;
      float timeout = sock->rtt_timer->timeout;
      time_t cur_time = time(NULL);
      time_t out_time = cur_time + timeout;
      // 初始化信号量
      while(time(NULL) < out_time) {
            if (sock->interrupt_signal == 1) {
                  printf("receive interrupt signal.\n");
                  // 更新RTT的值
                  tcp_ack_update_rtt(sock, time(NULL) - cur_time, 1);
                  sock->rtt_timer->chan = NULL;
                  return NULL;
            }else {
                  // 休息一会，防止不断轮询导致CPU负载过重
                  sleep(1);
            }
      }
      // 此时超时，调用回调函数
      sock->rtt_timer->callback(sock);
}
```

当收到 ACK 时，我们向计时器传递信号量并停止计时，重置计时器，否则调用回调函数。根据不同的 socket 状态进行处理：

```
// 当计时器超时时的回调函数
void tcp_write_timer_handler(tju_tcp_t* sock) {
	printf("timeout.\n");
	// 这里需要针对socket的状态进行不同的操作
	switch(sock->state) {
		case SYN_SENT:
			tcp_send_syn(sock);
		case SYN_RECV:
			tcp_send_syn_ack(sock);
		case ESTABLISHED:
			// 超时重传，这里或许需要判断一下重传的次数，若重传次数过多应该关闭连
接
			if(sock->timeout_counts > RETRANSMIT_LIMIT) {
				// 重传次数超限，关闭连接
				printf("重传次数超限，关闭连接.\n");
				tcp_outlimit_retransmit(sock);
			}else {
				// 重传分组
				sock->timeout_counts += 1;
				tcp_retransmit_timer(sock);
			}
		default:
			printf("Unresolved status.\n");
	}
}
```

涉及到的方法：

- `void tcp_init_timer(tju_tcp_t* sock, void (*retransmit_handler)(unsigned long))`: 初始化定时器并注册回调函数。 参数：待注册socket， 回调函数指针。 无返回值。
- `void tcp_init_rtt(tju_tcp_t* sock)`： 初始化RTT，仅仅在 `tcp_init_timer()` 中被调用。 参数： 待注册socket。 无返回值。
- `void tcp_set_estimator(tju_tcp_t* sock, float mrtt_us)`： 更新平滑RTT和RTT偏差值。 参数： 本地socket， 接收到ACK的RTT。 无返回值。
- `void tcp_bound_rto(tju_tcp_t* sock)`： 更新RTO，在 `tcp_set_estimator()` 后调用。 参数： 本地socket。 无返回值。
- `void tcp_set_rto(tju_tcp_t* sock)`： 仅仅调用 `tcp_bound_rto()`。
- `int tcp_ack_update_rtt(tju_tcp_t* sock, float seq_rtt_us, float sack_rtt_us)`： 收到 ACK 后更新 RTT，调用 `tcp_set_rto()` 和 `tcp_set_estimator()`。 参数：本地socket， 收到的RTT， 保底RTT。 成功返回0，失败返回-1。
- `void tcp_write_timer_handler(tju_tcp_t* sock)`：注册的回调函数，根据当前 socket 的不同状态进行处理，无返回值。

- `void tcp_start_timer(tju_tcp_t* sock)`: 开启定时器计时，无返回值。
- `void tcp_stop_timer(tju_tcp_t* sock)`：停止计时器计时并更新 RTT 的值，无返回值。
- `void tcp_retransmit_timer(tju_tcp_t* sock)`：超时重传函数处理，无返回值。
- `void tcp_entry_loss(tju_tcp_t* sock)`：超时慢启动处理，无返回值。
- `void tcp_outlimit_retransmit(tju_tcp_t* sock)`：超时重传次数过多，此时应当主动关闭连接。

4 流量控制

4.1 改进窗口算法

发送方根据接收方的rwnd,已接受的ack和发送的seq计算出接收方的useable window size, 如果其比值小于某个阈值则延迟发送新的数据，仅仅发送一个HEAD用于维持链接。

实现如下所示：

Tju_tcp中的 `void calculate_sending_buffer_depend_on_rwnd(tju_tcp_t* sock);` 根据rwnd计算出uwnd然后相应的计算出最大的可发送长度，然后调用 `sending_buffer_to_layer3(sock,sock->sending_len,TRUE);`

函数将可发送的最大长度的数据从`sending_buffer`中发送出去并且将最后一个段的push bit标记为 1 用于告知接收方该段为最后一个段。

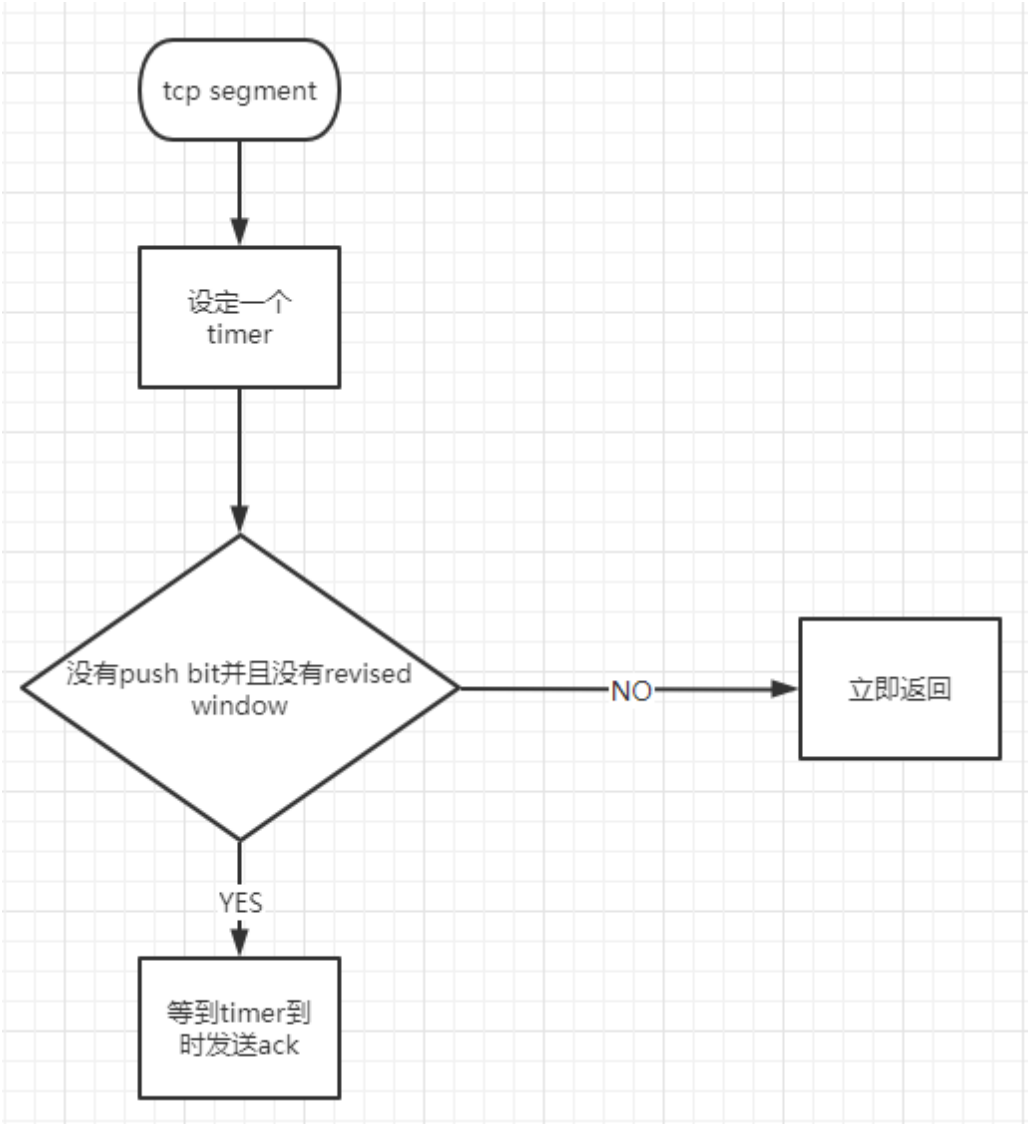
4.2 改善确认算法

前提：receiver 才能控制ACK 需要一个timer计数

当收到一个TCP段的时候，如果同时满足push bit 没有被设置并且没有revised window需要发回，那么就设置一个timer，间隔时间为大致为200ms~300ms，最合适的时间为自适应算法。

`void handle_delay_ack(tju_tcp_t* sock, char* pkt)`

在 `handle_TCP_PKT` 函数调用于控制ACK返回的延迟具体逻辑详见下图：



5 拥塞控制

5.1 Slow Start（慢启动）

（ssthresh：slow start thresh，慢启动门限值）

当cwnd的值小于 ssthresh 时，TCP 则处于 slow start 阶段，每收到一个 ACK，cwnd的值就会加1。

经过一个RTT的时间，cwnd的值就会变成原来的两倍，为指数增长。

5.2 Congestion Avoidance（拥塞避免）

当 cwnd 的值超过 ssthresh 时，就会进入 Congestion Avoidance 阶段，在该阶段下，cwnd以线性方式增长，大约每经过一个 RTT，cwnd 的值就会加1

5.3 Fast Retransmit（快重传）

按照拥塞避免算法中 cwnd 的增长趋势，迟早会造成拥塞（一般通过是否丢包来判断是否发生了拥塞）。

如果中网络中发生了丢包，通过等待一个 RTO 时间后再进行重传，是非常耗时的，因为RTO通常设置得会比较大（避免伪重传：不必要的重传）。

快重传的思想是：只要发送方收到了三个重复的 ACK，就会立马重传，而不用等到 RTO 到达（如果没有3个重复的 ACK 而包丢失了，就只能超时重传）；

并且将 ssthresh 的值设置为当前 cwnd 的一半，而cwnd减为1，重回slow start阶段。

5.4 Fast Recovery（快速恢复）算法。

当收到三个重复的 ACK 或是超过了 RTO 时间且尚未收到某个数据包的 ACK，Reno 就会认为丢包了，并认定网络中发生了拥塞。

Reno 会把当前的 ssthresh 的值设置为当前 cwnd 的一半，但是并不会回到 slow start 阶段，而是将cwnd设置为（更新后的）ssthresh+3MSS，之后cwnd呈线性增长。