# Computer Networks Lab3


## 黄嘉祺 221220108


# 1 Program Structure and Design

## 1.1 TCPSender

- 实验要求我们完成 `TCPSender` 类型，用于向TCP连接中的接收方发送数据并重发未被接收的数据，并接收ACK与cwnd以实现一定程度上的拥塞控制
- 具体来说，我们需要在 `src/tcp_sender.cc` 中实现 `sequence_numbers_in_flight()` 、 `consecutive_retransmissions()` 、 `push()` 、 `make_empty_message()` 、 `receive()` 、 `tick()` 这五个成员函数
  - `sequence_numbers_in_flight()` 用于返回当前已发送但还未被确认的TCP信息数量
  - `consecutive_retransmissions()` 用于返回当前连续重传的次数
  - `push()` 用于向TCP连接中发送数据，具体来说，它会首先考虑窗口中的剩余空间，然后从 `bytestream` 中读取尽可能多的数据装入 `TCPMessage` 中然后发送
  - `make_empty_message()` 用于生成一个空的 `TCPMessage` ，有时会用于发送 `ACK` 或 `RST` 等信息
  - `receive()` 用于接收来自接收方的 `TCPMessage` ，具体来说，它会首先检查是否收到 `RST` ，若收到则调用 `set_error` 函数将 `bytestream` 的 `error` 设为真，随后如果收到了合法的 `ACK` ，它将会检查缓存在未确认信息中的所有信息，将已经被确认的信息从未确认信息中删除，将未被确认的信息重新发送
  - `tick()` 用于模拟时间流逝，具体来说，它会首先检查是否有超时的信息，若有则将其重新发送，然后检查是否有信息需要重发，若需要则进行重发
- `sequence_numbers_in_flight()` 的实现思路是：
  - 扫描 `unack_buffer_` （缓存了所有未确认信息），并累加每条信息的长度，返回这个值

```cpp
uint64_t TCPSender::sequence_numbers_in_flight() const
{
  // Your code here.
  // return {};
  uint64_t res = 0;
  for ( auto msg : unack_buffer_ ) {
    res += msg.sequence_length();
  }
  return res;
}
```

- `consecutive_retransmissions()` 的实现思路是：

- 首先在 src/tcp_sender.hh 中定义了 uint64_t 类型的成员变量 retransmission_cnt ， 用于记录连续重传的次数
- 该函数会直接返回这个变量的值

```
uint64_t TCPSender::consecutive_retransmissions() const
{
  // Your code here.
  // return {};
  return retransmission_cnt;
}
```

- push() 的实现思路是：
  - 首先记录当前窗口大小，并减去未被确认信息的总长度，得到可用空间
  - 接下来通过循环不停地从 bytestream 中读取数据并发送，直到可用空间被占满或 bytestream 已没有数据（读到的数据长度为0）为止
  - 最后注意若 timer 未启动且此时仍有未确认信息，则启动 timer

```
void TCPSender::push( const TransmitFunction& transmit )
{
  // Your code here.
  // (void)transmit;
  uint64_t remain_space = ( windows_size_ == 0 ) ? 1 : windows_size_;
  if ( remain_space <= sequence_numbers_in_flight() ) {
    return;
  }
  remain_space -= sequence_numbers_in_flight();
  while ( remain_space > 0 ) {
    TCPSenderMessage message = make_empty_message();
    if ( !syn_sent_ ) {
      message.SYN = true;
      syn_sent_ = true;
      remain_space--;
    }
    message.seqno = Wrap32::wrap( first_unacked_, isn_ );
    read( input_.reader(), min( TCPConfig::MAX_PAYLOAD_SIZE, remain_space ), message.payload );
    remain_space -= message.payload.size();
    if ( !fin_sent_ && reader().is_finished() && remain_space > 0 ) {
      message.FIN = true;
      fin_sent_ = true;
      remain_space--;
    }
    if ( message.sequence_length() == 0 ) {
      break;
    }
    unack_buffer_.push_back( message );
    transmit( message );
    first_unacked_ += message.sequence_length();
  }
  if ( !timer_start_ && sequence_numbers_in_flight() != 0 ) {
    current_time_ = 0;
    current_RTO_ = initial_RTO_ms_;
    timer_start_ = true;
  }
}
```

- make_empty_message() 的实现思路是：
  - 首先生成一个空的 TCPMessage ， 然后将 message.seqno 设为 first_unacked （记得注意 wrap ） ， message.RST 设为 input_.has_error()
  - 最后返回这个 TCPMessage

```cpp
TCPSenderMessage TCPSender::make_empty_message() const
{
  // Your code here.
  // return {};
  TCPSenderMessage message;
  message.seqno = Wrap32::wrap( first_unacked_, isn_ );
  message.SYN = false;
  message.FIN = false;
  message.RST = input_.has_error();
  return message;
}
```

- `receive()` 的实现思路是：
  - 首先检查是否收到 `RST`，若收到则调用 `set_error` 函数将 `bytestream` 的 `error` 设为真
  - 将 `windows_size_` 设为 `msg.window_size` 指示的窗口长度
  - 接下来使用 `unwrap` 函数将收到的 `msg.ackno` 展开为 `received_ack`，然后扫描 `unack_buffer_`，将所有 `seqno` 小于 `received_ack` 的信息从 `unack_buffer_` 中删除
  - 最后若有从 `unack_buffer_` 中删除的信息，则将 `retransmission_cnt` 清零，并关闭 `timer`，将目前的 `RTO` 时间还原为初始值

```cpp
void TCPSender::receive( const TCPReceiverMessage& msg )
{
  // Your code here.
  // (void)msg;
  if ( msg.RST ) {
    input_.set_error();
  }
  windows_size_ = msg.window_size;
  if ( !msg.ackno.has_value() ) {
    return;
  }
  uint64_t received_ack = msg.ackno.value().unwrap( isn_, first_unacked_ );
  if ( received_ack > first_unacked_ ) {
    return;
  }
  bool fully_acked = false;
  for ( auto it = unack_buffer_.begin(); it != unack_buffer_.end(); it++ ) {
    if ( it->seqno.unwrap( isn_, first_unacked_ ) + it->sequence_length() <= received_ack ) {
      it = unack_buffer_.erase( it );
      it--;
      fully_acked = true;
    } else {
      break;
    }
  }
  if ( fully_acked ) {
    retransmission_cnt = 0;
    current_time_ = 0;
    timer_start_ = false;
    current_RTO_ = initial_RTO_ms_;
  }
}
```

- `tick()` 的实现思路是：
  - 首先根据 `ms_since_last_tick` 更新 `time_since_last_tick_`

- 然后若 timer 已启动且已超时，则将 unack_buffer 中最近一条被缓存的信息重发
- 若窗口大小大于0，则将 retransmission_cnt 加一，并将 RTO 时间翻倍
- 最后重新启动 timer

```
void TCPSender::tick( uint64_t ms_since_last_tick, const TransmitFunction& transmit )
{
  // Your code here.
  // (void)ms_since_last_tick;
  // (void)transmit;
  // (void)initial_RTO_ms_;
  current_time_ += ms_since_last_tick;
  if ( timer_start_ && current_time_ >= current_RTO_ ) {
    transmit( unack_buffer_.front() );
    if ( windows_size_ > 0 ) {
      retransmission_cnt++;
      current_RTO_ = current_RTO_ * 2;
    }
    current_time_ = 0;
    timer_start_ = true;
  }
}
```

# 2 Implementation Challanges

- 在实现 push 的时候，一开始在设置循环终止条件的时候只设置了当 remain_space 变为0的时候终止循环不再发送，忘了在循环中加一条判断 bytestream 中是否还有数据需要发送，导致测试 send_connect 时循环一直没有跳出，陷入死循环而超时，后加入一条判断
  若 message.sequnece_length() 为0则跳出循环，问题解决
- 最后发现如果测试时在 push 一条消息前先将 RST 设置为真的话，当 bytestream 被设置为 error 后会导致 read 函数无法正常从 bytestream 中读取，导致一直发送空信息，后经过仔细观察与分析发现是lab0在实现 bytestream 的 push 函数时错误地设置为了当 bytestream 已关闭或出现 error 时不读取直接返回空值，后将这个错误更正，问题解决

# 3 Remaining Bugs

目前我的代码已经通过了全部的测试，暂时未发现明显的bug

# 4 Experimental Results and Performance

- Bytestream, reassembler, wrapper, tcp receiver and tcp sender: Passed all 36 tests

```
      Start 23: recv_window
22/36 Test #23: recv_window .....................   Passed    0.02 sec
      Start 24: recv_reorder
23/36 Test #24: recv_reorder ....................   Passed    0.02 sec
      Start 25: recv_reorder_more
24/36 Test #25: recv_reorder_more ...............   Passed    1.66 sec
      Start 26: recv_close
25/36 Test #26: recv_close ......................   Passed    0.02 sec
      Start 27: recv_special
26/36 Test #27: recv_special ....................   Passed    0.03 sec
      Start 28: send_connect
27/36 Test #28: send_connect ....................   Passed    0.02 sec
      Start 29: send_transmit
28/36 Test #29: send_transmit ...................   Passed    0.89 sec
      Start 30: send_retx
29/36 Test #30: send_retx .......................   Passed    0.02 sec
      Start 31: send_window
30/36 Test #31: send_window .....................   Passed    0.13 sec
      Start 32: send_ack
31/36 Test #32: send_ack ........................   Passed    0.02 sec
      Start 33: send_close
32/36 Test #33: send_close ......................   Passed    0.02 sec
      Start 34: send_extra
33/36 Test #34: send_extra ......................   Passed    0.06 sec
      Start 37: compile with optimization
34/36 Test #37: compile with optimization ........   Passed    0.08 sec
      Start 38: byte_stream_speed_test
             ByteStream throughput: 2.63 Gbit/s
35/36 Test #38: byte_stream_speed_test ...........   Passed    0.10 sec
      Start 39: reassembler_speed_test
             Reassembler throughput: 6.47 Gbit/s
36/36 Test #39: reassembler_speed_test ...........   Passed    0.16 sec

100% tests passed, 0 tests failed out of 36

Total Test time (real) =   6.70 sec
Built target check3
```