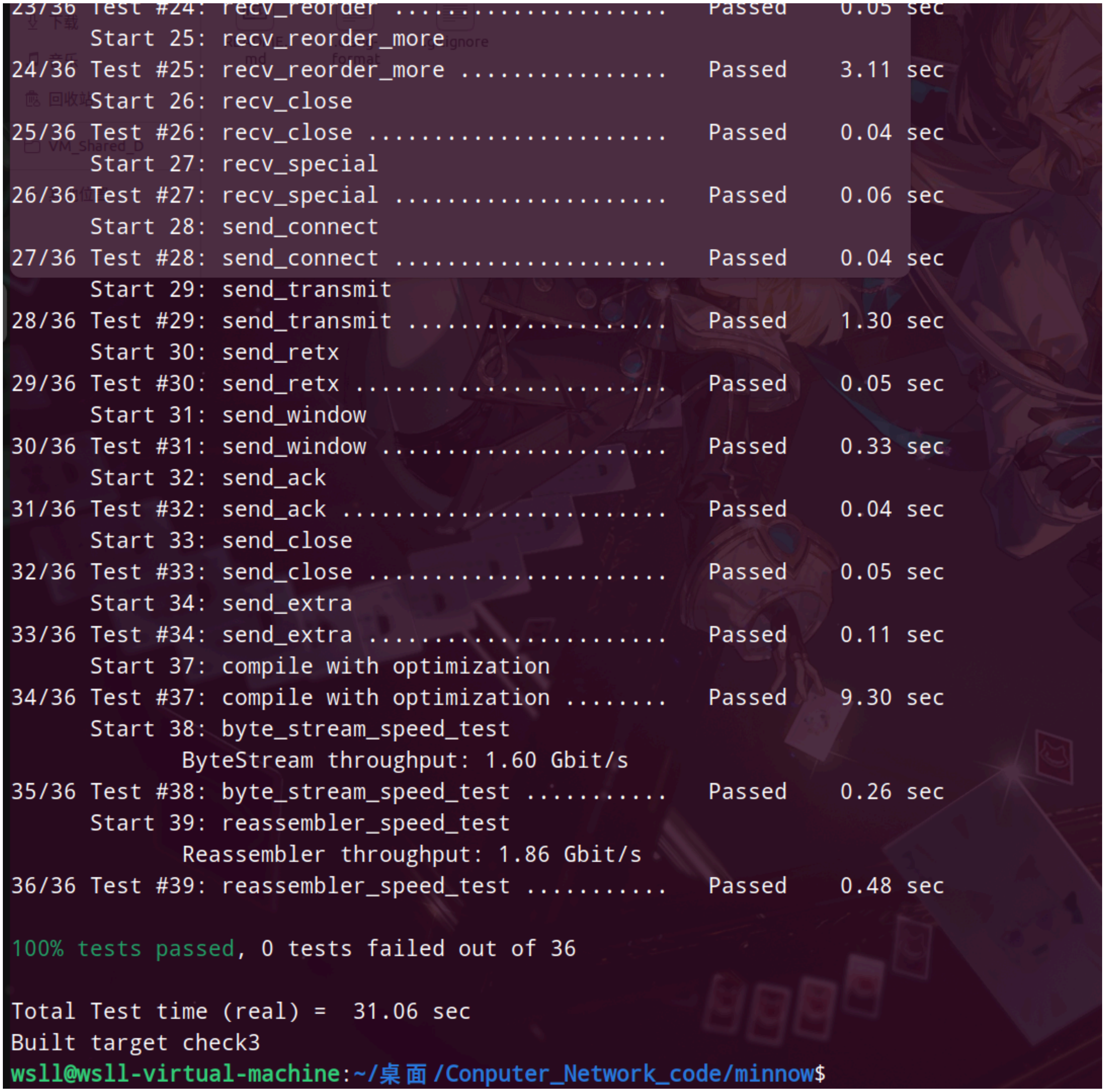
计网lab3

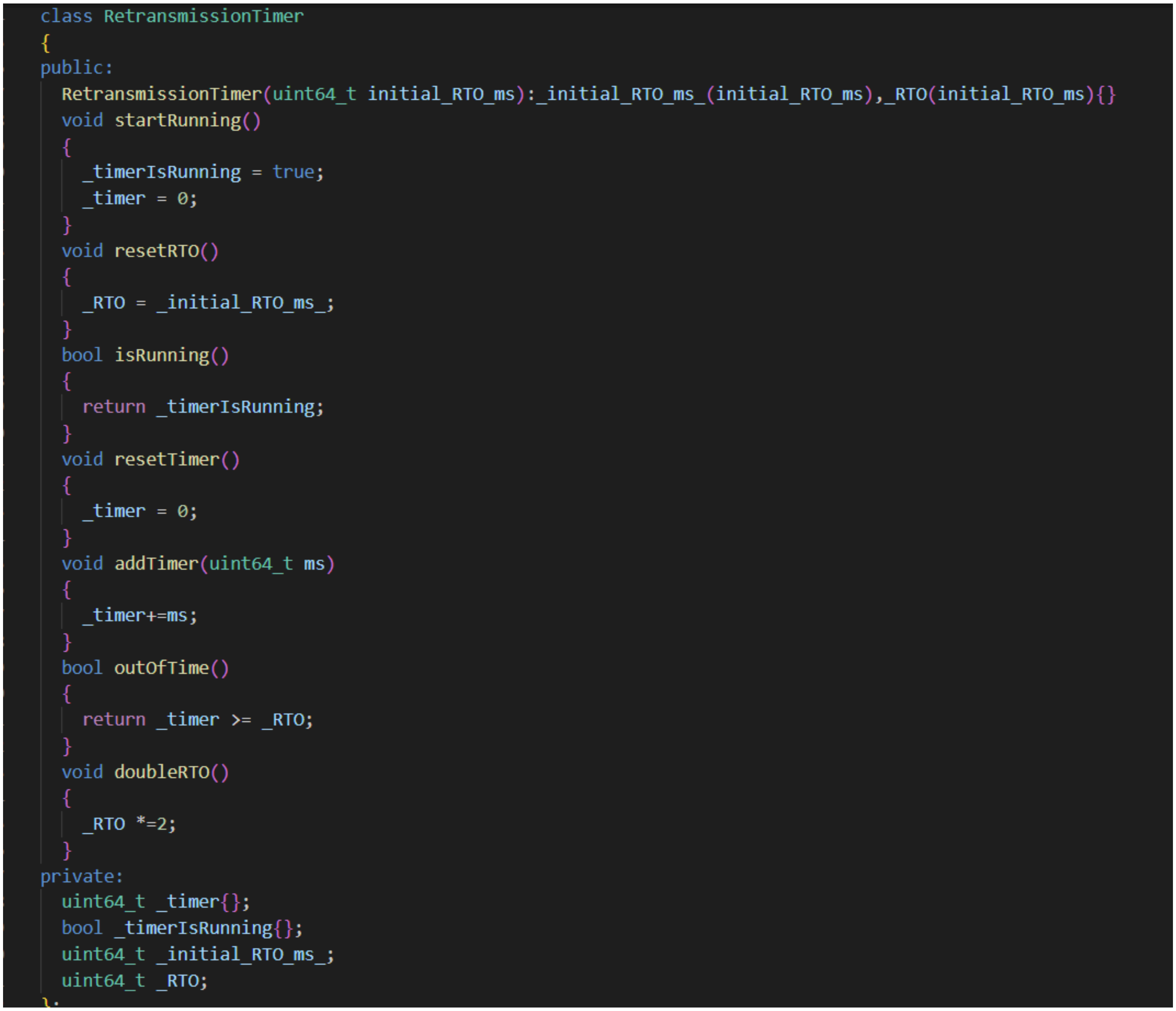
最终结果：



```
23/36 Test #24: recv_reorder .....................   Passed    0.05 sec
      Start 25: recv_reorder_more
24/36 Test #25: recv_reorder_more ................   Passed    3.11 sec
      Start 26: recv_close
25/36 Test #26: recv_close .......................   Passed    0.04 sec
      Start 27: recv_special
26/36 Test #27: recv_special .....................   Passed    0.06 sec
      Start 28: send_connect
27/36 Test #28: send_connect .....................   Passed    0.04 sec
      Start 29: send_transmit
28/36 Test #29: send_transmit ....................   Passed    1.30 sec
      Start 30: send_retx
29/36 Test #30: send_retx ........................   Passed    0.05 sec
      Start 31: send_window
30/36 Test #31: send_window ......................   Passed    0.33 sec
      Start 32: send_ack
31/36 Test #32: send_ack .........................   Passed    0.04 sec
      Start 33: send_close
32/36 Test #33: send_close .......................   Passed    0.05 sec
      Start 34: send_extra
33/36 Test #34: send_extra .......................   Passed    0.11 sec
      Start 37: compile with optimization
34/36 Test #37: compile with optimization ........   Passed    9.30 sec
      Start 38: byte_stream_speed_test
             ByteStream throughput: 1.60 Gbit/s
35/36 Test #38: byte_stream_speed_test ..........   Passed    0.26 sec
      Start 39: reassembler_speed_test
             Reassembler throughput: 1.86 Gbit/s
36/36 Test #39: reassembler_speed_test ..........   Passed    0.48 sec

100% tests passed, 0 tests failed out of 36

Total Test time (real) =  31.06 sec
Built target check3
wsll@wsll-virtual-machine:~/桌面/Conputer_Network_code/minnow$
```

设计思路：

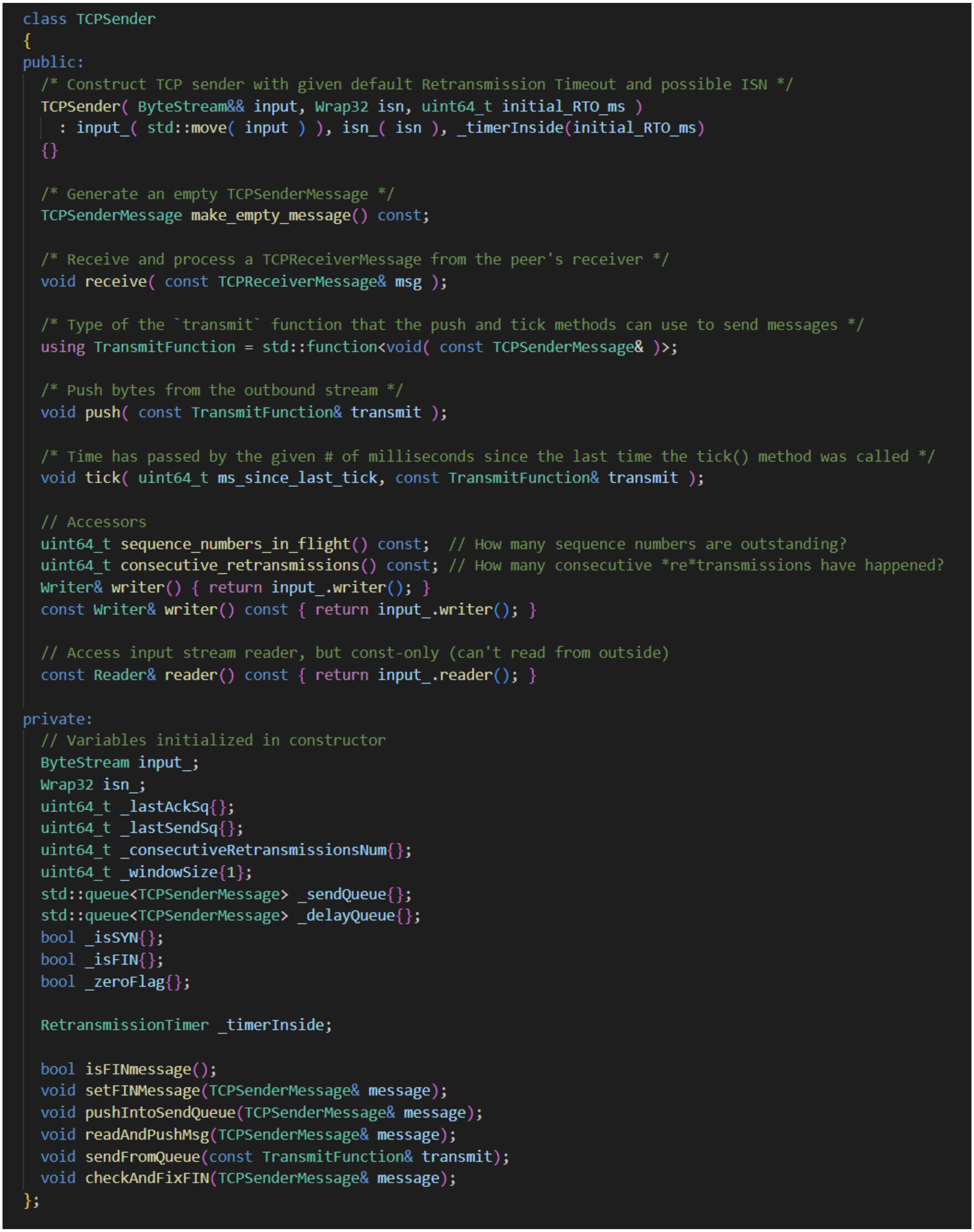
计时器类



```
class RetransmissionTimer
{
public:
  RetransmissionTimer(uint64_t initial_RTO_ms):_initial_RTO_ms_(initial_RTO_ms),_RTO(initial_RTO_ms){}
  void startRunning()
  {
    _timerIsRunning = true;
    _timer = 0;
  }
  void resetRTO()
  {
    _RTO = _initial_RTO_ms_;
  }
  bool isRunning()
  {
    return _timerIsRunning;
  }
  void resetTimer()
  {
    _timer = 0;
  }
  void addTimer(uint64_t ms)
  {
    _timer+=ms;
  }
  bool outOfTime()
  {
    return _timer >= _RTO;
  }
  void doubleRTO()
  {
    _RTO *=2;
  }
private:
  uint64_t _timer{};
  bool _timerIsRunning{};
  uint64_t _initial_RTO_ms_;
  uint64_t _RTO;
};
```

依据文档，我们实现计时器几个基础的功能，并且实现封装。

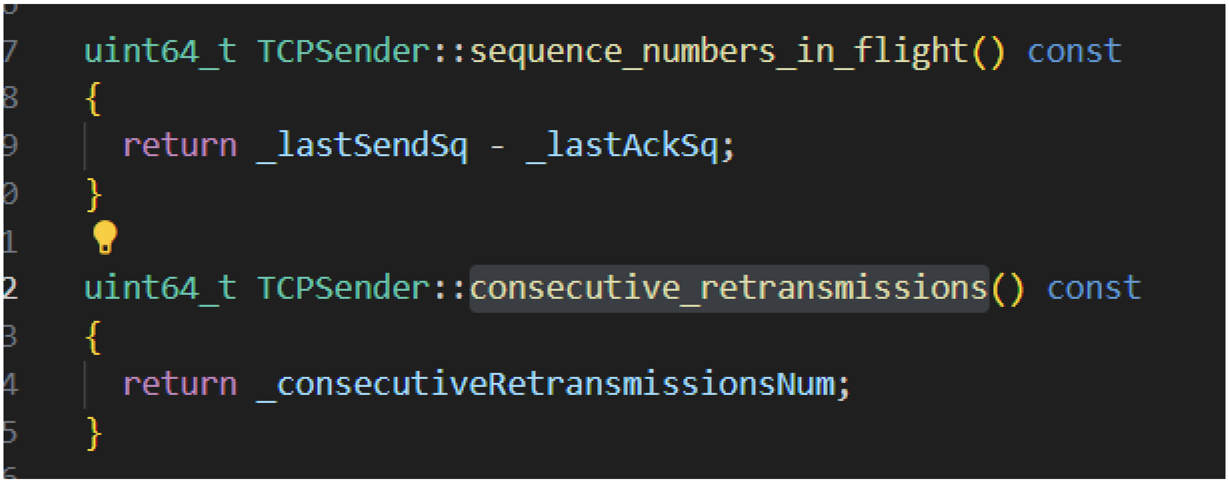
TCPSender类



```
class TCPSender
{
public:
  /* Construct TCP sender with given default Retransmission Timeout and possible ISN */
  TCPSender( ByteStream&& input, Wrap32 isn, uint64_t initial_RTO_ms )
    : input_( std::move( input ) ), isn_( isn ), _timerInside(initial_RTO_ms)
  {}

  /* Generate an empty TCPSenderMessage */
  TCPSenderMessage make_empty_message() const;

  /* Receive and process a TCPReceiverMessage from the peer's receiver */
  void receive( const TCPReceiverMessage& msg );

  /* Type of the `transmit` function that the push and tick methods can use to send messages */
  using TransmitFunction = std::function<void( const TCPSenderMessage& )>;

  /* Push bytes from the outbound stream */
  void push( const TransmitFunction& transmit );

  /* Time has passed by the given # of milliseconds since the last time the tick() method was called */
  void tick( uint64_t ms_since_last_tick, const TransmitFunction& transmit );

  // Accessors
  uint64_t sequence_numbers_in_flight() const;  // How many sequence numbers are outstanding?
  uint64_t consecutive_retransmissions() const; // How many consecutive *re*transmissions have happened?
  Writer& writer() { return input_.writer(); }
  const Writer& writer() const { return input_.writer(); }

  // Access input stream reader, but const-only (can't read from outside)
  const Reader& reader() const { return input_.reader(); }

private:
  // Variables initialized in constructor
  ByteStream input_;
  Wrap32 isn_;
  uint64_t _lastAckSq{};
  uint64_t _lastSendSq{};
  uint64_t _consecutiveRetransmissionsNum{};
  uint64_t _windowSize{1};
  std::queue<TCPSenderMessage> _sendQueue{};
  std::queue<TCPSenderMessage> _delayQueue{};
  bool _isSYN{};
  bool _isFIN{};
  bool _zeroFlag{};

  RetransmissionTimer _timerInside;

  bool isFINmessage();
  void setFINMessage(TCPSenderMessage& message);
  void pushIntoSendQueue(TCPSenderMessage& message);
  void readAndPushMsg(TCPSenderMessage& message);
  void sendFromQueue(const TransmitFunction& transmit);
  void checkAndFixFIN(TCPSenderMessage& message);
};
```

添加了一些private成员和函数。
_lastAckSq 和 _lastSendSq 用来维护最近发送的序列号和已经ACK的最大序列号。
_consecutiveRetransmissionsNum 用来维护连续重传的数量。
_windowSize 用来维护窗口大小。
_zeroFlag 用来辨别“full”的窗口和“zero-size”的窗口。
_sendQueue： push的时候会入队列。
_delayQueue： 已发送的消息会在当中暂存，以便于超时重传。
_timerInside： 内置的计时器类。

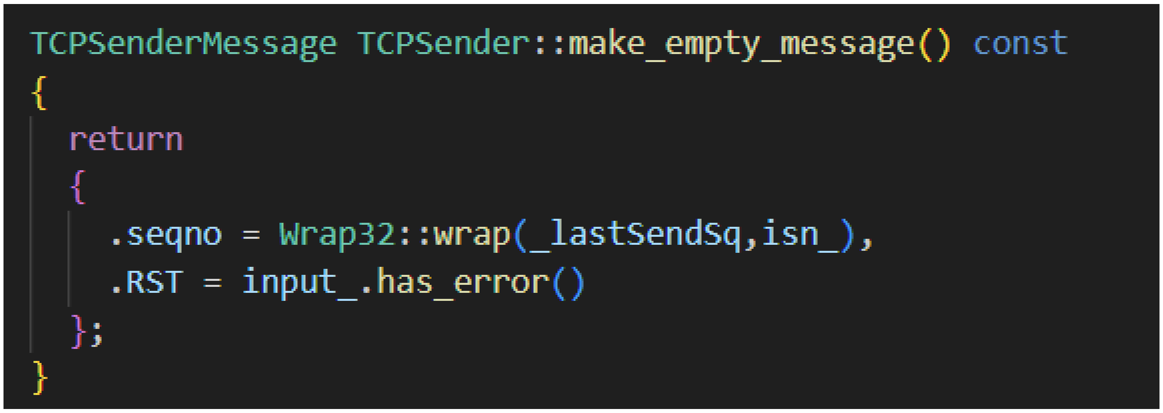
sequence_numbers_in_flight 和 consecutive_retransmissions



```
uint64_t TCPSender::sequence_numbers_in_flight() const
{
  return _lastSendSq - _lastAckSq;
}

uint64_t TCPSender::consecutive_retransmissions() const
{
  return _consecutiveRetransmissionsNum;
}
```

正常依靠私有成员维护即可。

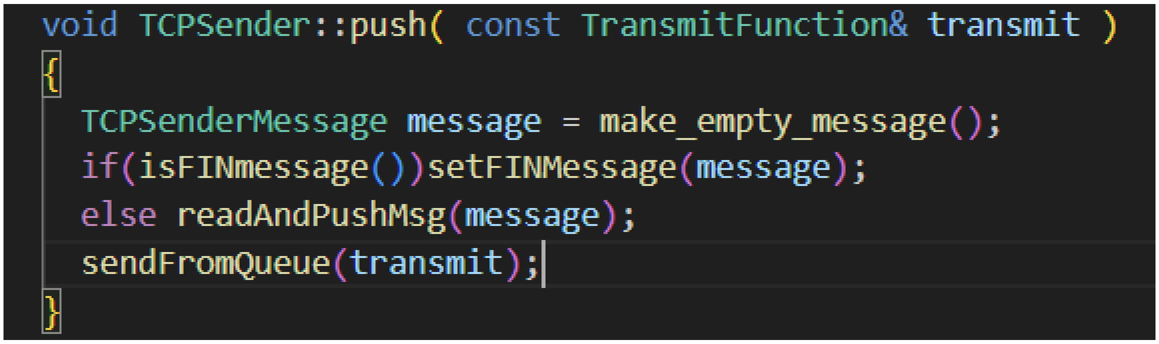
make_empty_message



```
TCPSenderMessage TCPSender::make_empty_message() const
{
  return
  {
    .seqno = Wrap32::wrap(_lastSendSq,isn_),
    .RST = input_.has_error()
  };
}
```

使用指派初始化器进行初始化。这里FIN和SYN等关键数据会被默认设置为false。

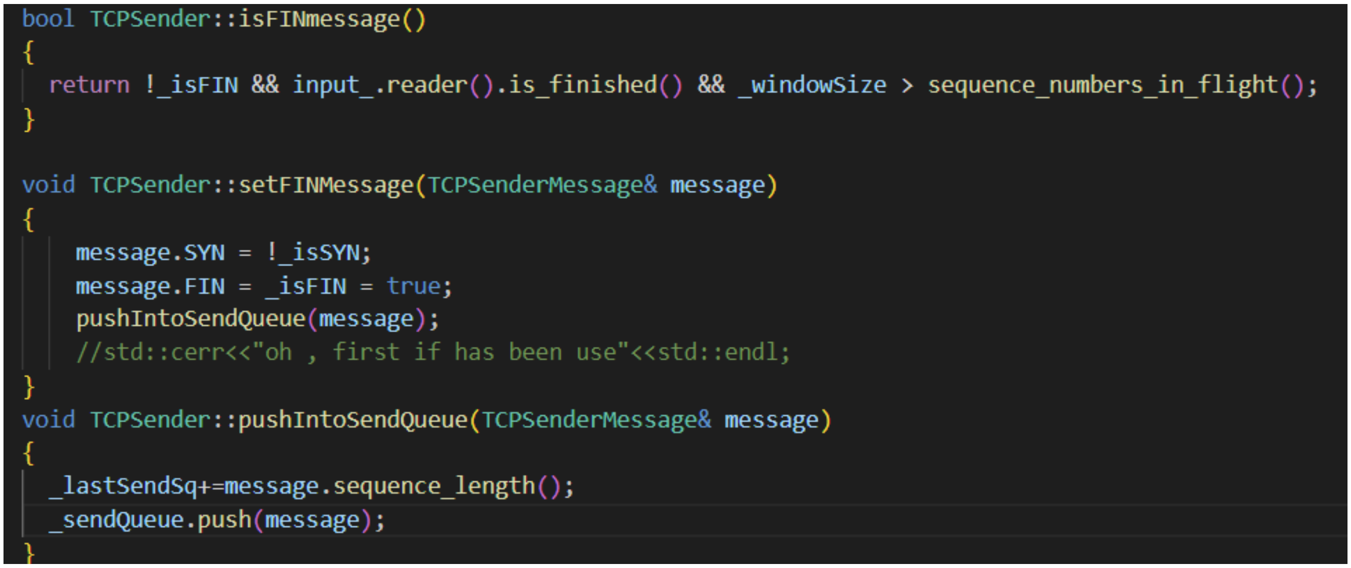
push



```
void TCPSender::push( const TransmitFunction& transmit )
{
  TCPSenderMessage message = make_empty_message();
  if(isFINmessage())setFINMessage(message);
  else readAndPushMsg(message);
  sendFromQueue(transmit);
}
```

我们把要发送的分成是FIN还是非FIN两类，并且最后统一从队列中进行发送。

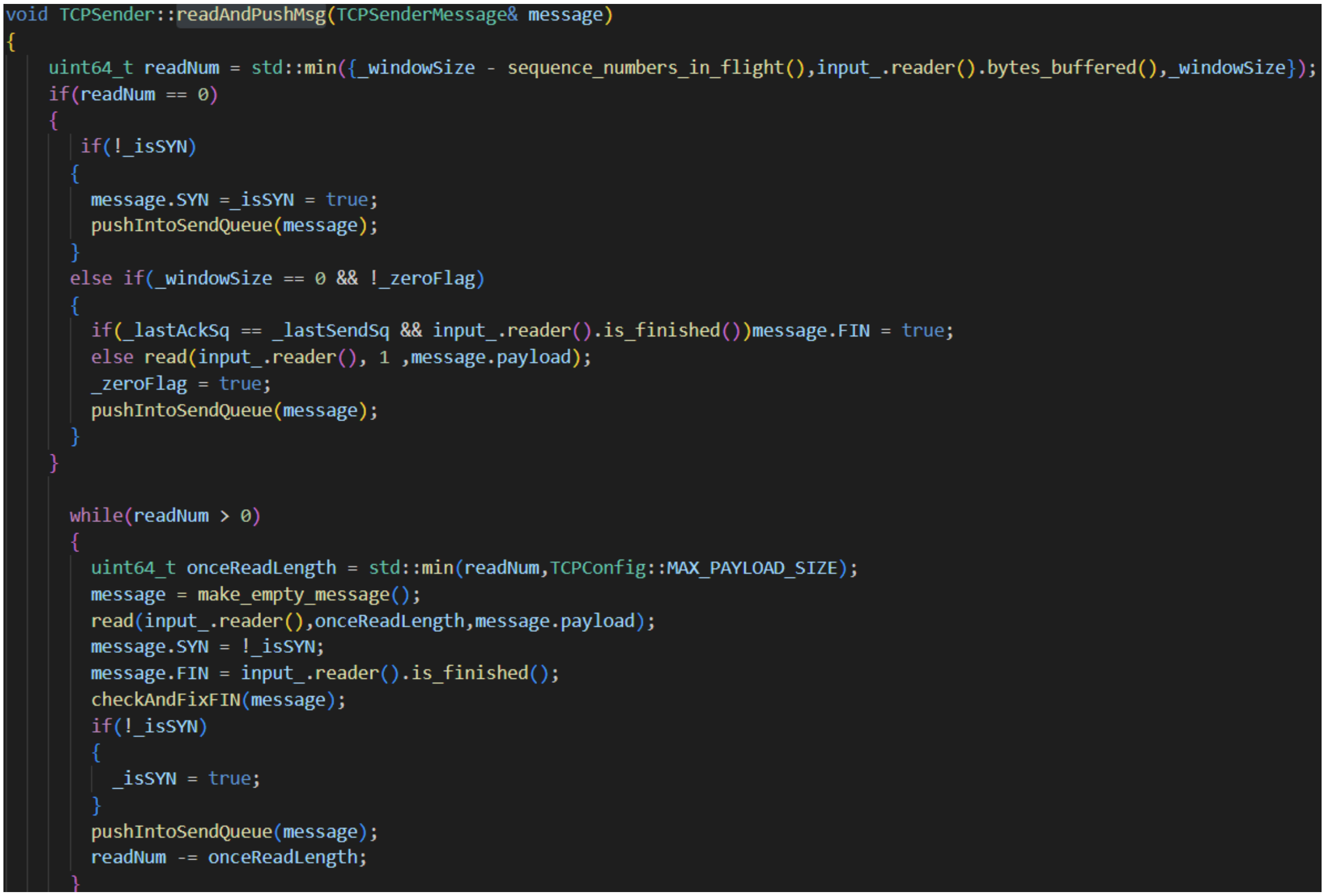
isFINmessage 和 setFINMessage



```
bool TCPSender::isFINmessage()
{
  return !_isFIN && input_.reader().is_finished() && _windowSize > sequence_numbers_in_flight();
}

void TCPSender::setFINMessage(TCPSenderMessage& message)
{
    message.SYN = !_isSYN;
    message.FIN = _isFIN = true;
    pushIntoSendQueue(message);
    //std::cerr<<"oh , first if has been use"<<std::endl;
}
void TCPSender::pushIntoSendQueue(TCPSenderMessage& message)
{
  _lastSendSq+=message.sequence_length();
  _sendQueue.push(message);
}
```

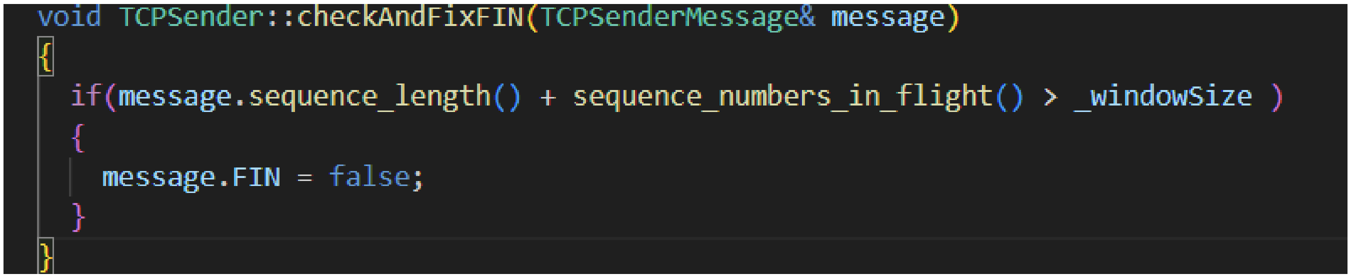
注：这里的FIN如果超过了窗口大小，我们在后续会进行裁剪。

readAndPushMsg



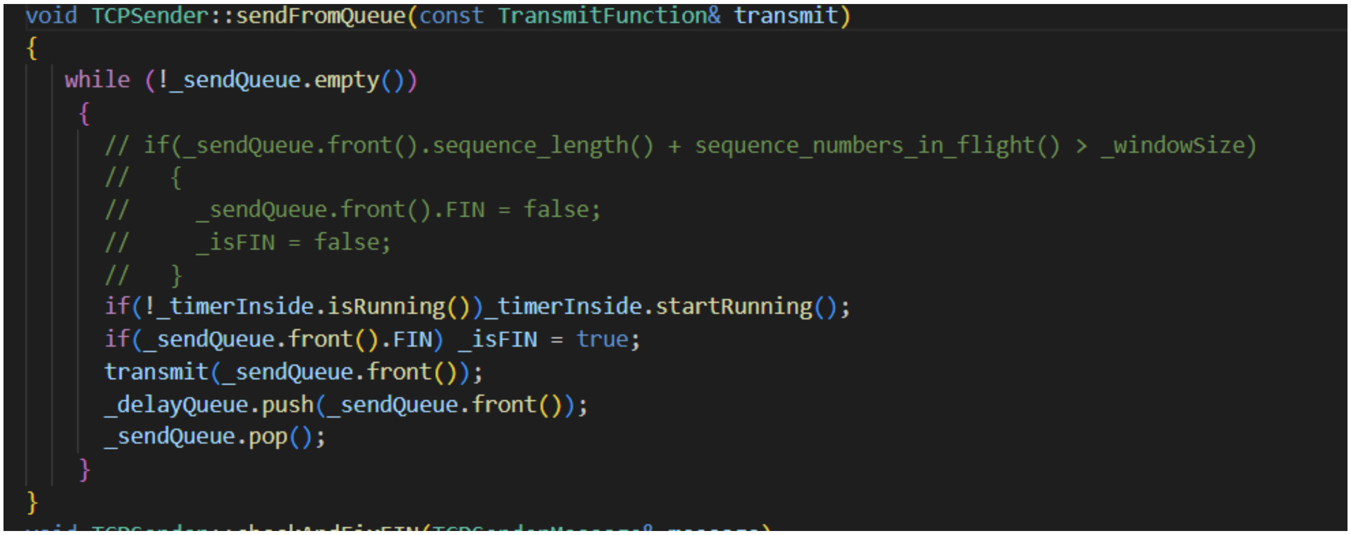
```
void TCPSender::readAndPushMsg(TCPSenderMessage& message)
{
    uint64_t readNum = std::min({_windowSize - sequence_numbers_in_flight(),input_.reader().bytes_buffered(),_windowSize});
    if(readNum == 0)
    {
      if(!_isSYN)
      {
        message.SYN = _isSYN = true;
        pushIntoSendQueue(message);
      }
      else if(_windowSize == 0 && !_zeroFlag)
      {
        if(_lastAckSq == _lastSendSq && input_.reader().is_finished())message.FIN = true;
        else read(input_.reader(), 1 ,message.payload);
        _zeroFlag = true;
        pushIntoSendQueue(message);
      }
    }

      while(readNum > 0)
      {
        uint64_t onceReadLength = std::min(readNum,TCPConfig::MAX_PAYLOAD_SIZE);
        message = make_empty_message();
        read(input_.reader(),onceReadLength,message.payload);
        message.SYN = !_isSYN;
        message.FIN = input_.reader().is_finished();
        checkAndFixFIN(message);
        if(!_isSYN)
        {
          _isSYN = true;
        }
        pushIntoSendQueue(message);
        readNum -= onceReadLength;
      }
```

先判别需要read的字节数，这需要在窗口可容纳的大小和缓冲区大小中取最小值。另外我们特别需要注意减成溢出的溢出，因此需要把windowSize的大小考虑上。
如果需要read的字节数是0，那么有可能是没开始传输也可能是出现了文档中的0窗口的情况，我们针对两种情况处理message的标志位。
最后进行一般情况的处理，依据文档，把TCPConfig::MAX_PAYLOAD_SIZE考虑上。特别需要注意裁剪FIN的值。
裁剪的函数如下：



```
void TCPSender::checkAndFixFIN(TCPSenderMessage& message)
{
  if(message.sequence_length() + sequence_numbers_in_flight() > _windowSize )
  {
    message.FIN = false;
  }
}
```

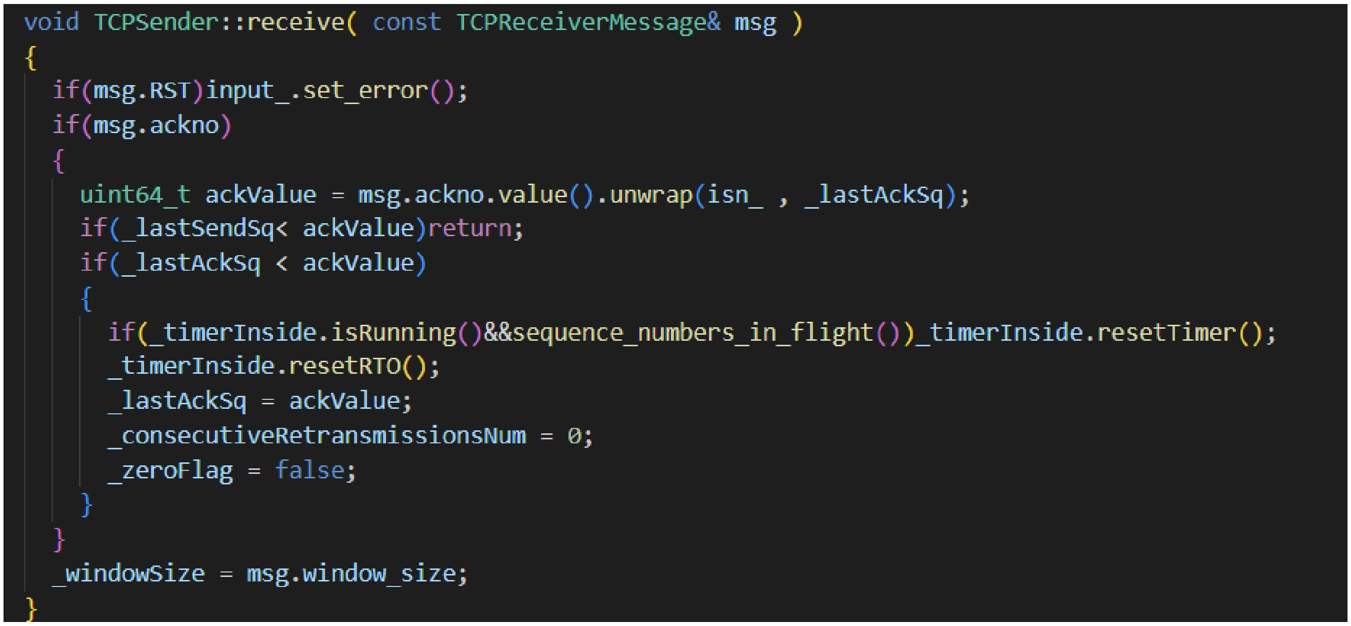
sendFromQueue



```
void TCPSender::sendFromQueue(const TransmitFunction& transmit)
{
   while (!_sendQueue.empty())
    {
      // if(_sendQueue.front().sequence_length() + sequence_numbers_in_flight() > _windowSize)
      //   {
      //     _sendQueue.front().FIN = false;
      //     _isFIN = false;
      //   }
      if(!_timerInside.isRunning())_timerInside.startRunning();
      if(_sendQueue.front().FIN) _isFIN = true;
      transmit(_sendQueue.front());
      _delayQueue.push(_sendQueue.front());
      _sendQueue.pop();
    }
}
void TCPSender::checkAndFixFIN(TCPSenderMessage& message)
```

依次对队列中的message进行发送。另外在未启动时注意timer的启动。

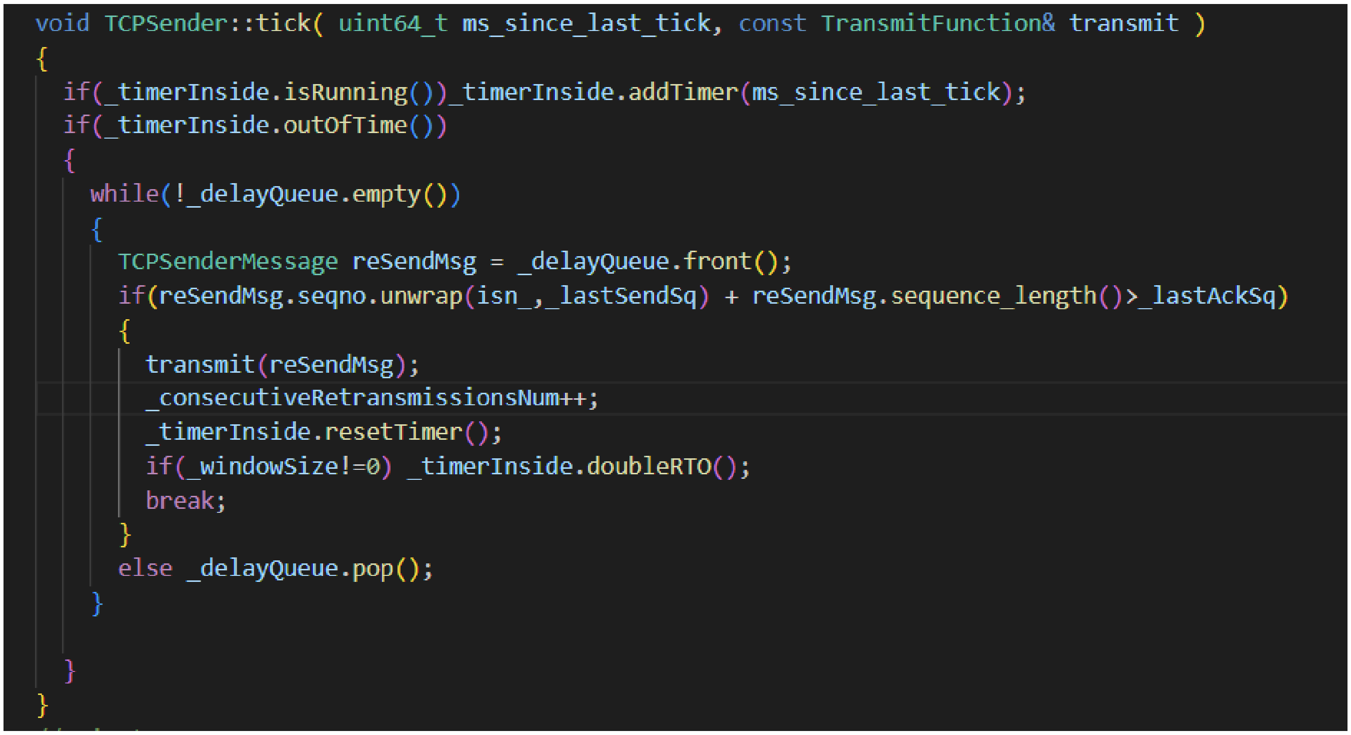
receive



```
void TCPSender::receive( const TCPReceiverMessage& msg )
{
  if(msg.RST)input_.set_error();
  if(msg.ackno)
  {
    uint64_t ackValue = msg.ackno.value().unwrap(isn_ , _lastAckSq);
    if(_lastSendSq< ackValue)return;
    if(_lastAckSq < ackValue)
    {
      if(_timerInside.isRunning()&&sequence_numbers_in_flight())_timerInside.resetTimer();
      _timerInside.resetRTO();
      _lastAckSq = ackValue;
      _consecutiveRetransmissionsNum = 0;
      _zeroFlag = false;
    }
  }
  _windowSize = msg.window_size;
}
```

如果optional有值，那么就进行处理。如果收到的ack比之前发送过的还大，那么说明出了问题，直接丢弃；如果是正常的，那么处理timer，更新状态数据。

tick



```
void TCPSender::tick( uint64_t ms_since_last_tick, const TransmitFunction& transmit )
{
  if(_timerInside.isRunning())_timerInside.addTimer(ms_since_last_tick);
  if(_timerInside.outOfTime())
  {
    while(!_delayQueue.empty())
    {
      TCPSenderMessage reSendMsg = _delayQueue.front();
      if(reSendMsg.seqno.unwrap(isn_,_lastSendSq) + reSendMsg.sequence_length()>_lastAckSq)
      {
        transmit(reSendMsg);
        _consecutiveRetransmissionsNum++;
        _timerInside.resetTimer();
        if(_windowSize!=0) _timerInside.doubleRTO();
        break;
      }
      else _delayQueue.pop();
    }

  }
}
```

如果出于running的状态，那么需要把计时器加上过去的事件。
如果超时了，那么对之前的延时队列检索，寻找第一个序列号大于ack的消息，重发。更新状态和RTO（特别注意windowsize是0的情况），并且重新计时即可。

遇到的困难和改进思考

许多边界条件都需要依据测试用例来进行反推处理。
目前传输效率仍处于正常的状态。