浙江大学

本科实验报告

课程名称：	计算机网络基础
实验名称：	Lab4 the TCP receiver and the TCP sender
姓　　名：	蔡佳伟
学　　院：	计算机学院
系：	计算机科学与技术学院
专　　业：	软件工程
学　　号：	3220104519
指导教师：	高艺

2024 年 11 月 25 日

浙江大学实验报告

实验名称：Lab4 the TCP receiver and the TCP sender 实验类型：设计实验

同组学生：_____ 实验地点：_____

一、 实验目的：

- 学习掌握 TCP 的工作原理
- 学习掌握流重组器的工作原理
- 学习掌握 TCP receiver 和 TCP sender 的相关知识

二、 实验内容

- 实现流重组器，一个将字节流的字串或小段按照正确顺序来拼接回连续字节流的模块。
- 实现 TCPReceiver。
  - 接收 TCPsegment；
  - 重新组装字节流；
  - 确定应该发回发送者的信号，以进行数据确认和流量控制。
- 实现 TCPSender：
  - 将 ByteStream 中的数据以 TCP 报文形式持续发送给接收者；
  - 处理 TCPReceiver 传入的 ackno 和 window size，以追踪接收者当前的接收状态，以及检测丢包的情况；
  - 若经过一个超时时间仍然没有接收到 TCPReceiver 发送的 ack 包，则重传。

三、 主要仪器设备

- 联网的 PC 机
- Linux 虚拟机

四、 操作方法与实验步骤

- 实现流重组器：流重组器将网络中乱序的数据报重新组合成开始时的连续字节流，接收由一串字节组成的子字符串，以及该字符串的第一个字节在更大的流中的索引。
  - 实现 StreamReassembler 接口，即 **stream_reassembler.hh** 文件中的主要方法：

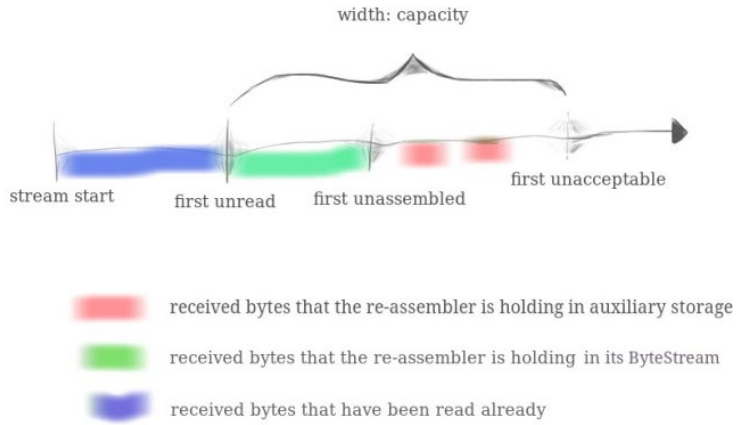
```
// `data`: the substring
// `index` indicates the index (place in sequence) of the first byte in `data`
// `eof`: the last byte of this substring will be the last byte in the entire stream
void push_substring(const string &data, const uint64_t index, const bool eof);

// Access the reassembled ByteStream (your code from Lab 0)
ByteStream &stream_out();

// The number of bytes in the substrings stored but not yet reassembled
size_t unassembled_bytes() const;
```

注意：

- **push_substring** 方法会忽略字符串中会导致 **StreamReassembler** 超过“capacity”的部分，下面这张图有助于你更好地理解流重组器。



- 为了减少工作量，流重组器的代码在 webget 实验拉取下来的代码中已给出，不会影响前面实验的测试，请务必理解这些代码，因为后续对 sender 和 receiver 需要用到流重组器，当然如果你有更好的方法可以重新实现这部分代码。

- 处理索引问题：在开始实现 receiver 和 sender 前，你需要先处理索引的问题。在流重组器中，每个字节都有一个 64 位的流索引，流中的第一个字节索引总是 0。64 位索引足够大，我们可以将其视为不会溢出。然而在 TCP 报头中，流中的每个字节索引不是用 64 位索引表示的，而是用 32 位的序列号表示。这增加了三个复杂性：

  1. 你需要考虑到 32 位的索引的大小问题：通过 TCP 发送的字节流长度往往没有限制，2^32 不是很大。一旦 32 位的序列号计数到 2^32 - 1，流中下个自己的序列号将为 0。

  2. TCP 序列号从一个随机值开始：TCP 试图确保序列号不会被猜测，也不太可能重复。所以流的序列号不是从零开始的。流中的第一个序列号是一个 32 位的随机数字，称为初始序列号(ISN)。这是表示 SYN(流的开始)的序列号。其余的序列号在此之后增加：数据的第一个字节将具有 ISN+1 的序列号(mod 232)，第二个字节将具有 ISN+2 (mod 232)...

  3. 逻辑开始和结束各自占用一个序列号：除了确保接收所有数据字节外，TCP 还确保可靠地接收流的开始和结束。因此，在 TCP 中，SYN(流开始)和 FIN(流结束)控制标志也被分配序列号。SYN 标志占用的序列号是 ISN，流中的每个字节数据也占用一个序列号。SYN 和 FIN 不是流本身的一部分，也不是“字节”—它们代表字节流本身的开始和结束。

- 序列号在每个 TCP 段的报头中传输，需要梳理绝对序列号和流索引的关系，你可以通过下面这张图来理解：

element	syn	c	a	t	fin
seqno	$2^{32}-2$	$2^{32}-1$	0	1	2
absolute seqno	0	1	2	3	4
stream index		0	1	2	

- 绝对序列号和流索引之间的转换只需加减 1 即可，但是序列号和绝对序列号之间的转换有点困难，混淆两者可能会产生棘手的错误。为了防止这些错误，我们将用自定义类型 **WrappingInt32** 表示序列号，并编写它和绝对序列号（用 **uint64_t** 表示）之间的转换，实现 **wrapping_intergers.hh** 中提供的方法：

1. `WrappingInt32 wrap(uint64_t n, WrappingInt32 isn)`

- 给定一个绝对序列号（n）和初始序列号（isn），生成序列号。

2. `uint64_t unwrap(WrappingInt32 n, WrappingInt32 isn, uint64_t checkpoint)`

- 给定序列号（n）、初始序列号（isn）和检查点绝对序列号，计算离检查点最近的对应的绝对序列号。

- 完成代码编写后，从 build 目录运行 **ctest -R wrap** 命令对 **WrappingInt32** 进行测试。

- 实现 TCPReceiver: receiver 的主要功能在上面已经给出。

- 在开始实现 receiver 的代码前，请复习 TCPSegment 和 TCPHeader 的文档。你可能对 length_in_sequence_space()方法感兴趣，该方法计算一个段占用多少序列号（包括 SYN 和 FIN 标志的序列号）。

- 实现 TCPReceiver 接口中的方法，下面给出的只是主要方法。

```
// Handle an inbound TCP segment
void segment_received(const TCPSegment &seg);
```

- 每次从对端接收到新的段时，都会调用该方法。

- 如果需要，设置初始序列号。设置了 SYN 标志的第一个到达的网段的序列号设置为 ISN。

- 推送任何数据到 **StreamReassembler**，包括流结束标记。如果 FIN 标志设置在 **TCPSegment** 的报头中，这意味着有效负载的最后一个字节是整个流的最后一个字节。

```
// The ackno that should be sent to the peer
//
// returns empty if no SYN has been received
//
// This is the beginning of the receiver's window, or in other words,
// the sequence number of the first byte in the stream
// that the receiver hasn't received.
std::optional<WrappingInt32> ackno() const;
```

- 返回一个 optional<WrappingInt32>，其中包含接收端不知道的第一个字节的序列号。这是窗口的左边界：接收端希望接收的第一个字节。如果 ISN 还没有被设置，则返回空。

```
// The window size that should be sent to the peer
//
// Formally: this is the size of the window of acceptable indices
// that the receiver is willing to accept. It's the distance between
// the ``first unassembled'' and the ``first unacceptable'' index.
//
// In other words: it's the capacity minus the number of bytes that the
// TCPReceiver is holding in the byte stream.
size_t window_size() const;
```

- 返回 **first_unassembled** 索引以及 **first_unacceptable** 索引。

- 有关 TCPSender 和 TCPReceiver 的代码主要涉及到文件如下(libsponge 目录下)：
  - tcp_sender.hh(sender 接口)
  - tcp_sender.cc(sender 接口的实现)
  - tcp_receiver.hh(reader 接口)
  - tcp_receiver.cc(reader 接口的实现)
- 实现 TCPSender: sender 的主要功能已在上面给出。TCPSender 和 TCPReceiver 各自负责 TCP 段的一部分。TCPSender 写入 TCPSegment 中与 TCPReceiver 相关的所有字段:即序列号、SYN 标志、有效负载和 FIN 标志。然而，TCP 发送方只读取由接收方写入的段中的字段:确认号和窗口大小。

  - 在开始代码实现前，这里有一些关于超时重传的描述，请仔细阅读：
    1. 每隔几毫秒，TCPSender 中的 tick 方法会被调用，并带有一个参数，该参数告诉它自上次调用该方法以来已经过了多少毫秒，使用它来维护 TCPSender 存在的总毫秒数的概念。
    2. 当 TCPSender 被构造时，会给它一个 参数，告诉它超时重传（RTO）的初始值，初始值保存在名为_initial_retransmission_timeout 的变量中。
    3. 你将实现一个重传计时器，在特定时间启动，一旦 RTO 结束，警报就会响起。
    4. 当所有数据都被确认后，停止重传计时器。
    5. 当 tick 方法被调用并且已经到了重传时间：
       - 重传接收端尚未完全确认的最早的段，你需要将未完成的段存储在一些内部数据结构中
       - 如果窗口大小为非零：
       - 跟踪连续重传的数量，并增加它。TCPConnection 将使用此信息来决定连接是否没有希望（连续多次重传），是否需要终止。
       - 将 RTO 的值翻倍。这被称为“指数后退”，它减慢了糟糕网络上的重传速度，以避免进一步的混乱。
       - 重置重传计时器并启动它，使它在 RTO 毫秒过期。
    6. 当接收端给发送端一个确认成功接收到新数据的 **ackno** 时：
       - 将 RTO 重置为初始值。
       - 如果发送方有任何未完成的数据，重新启动重传计时器。
       - 将连续重传次数重置为零。
  - 实现 TCPSender 接口中的方法：

    1. void fill_window()

    - TCPSender 需要填充窗口，将输入的 ByteStream 读取并以 TCPSegment 的形

式尽可能发送多的字节。

- 你需要确保发送的每个 TCPSegment 都完全适合接收方的窗口，使每个 TCPSegment 尽可能大，但不要超过 TCPConfig::MAX_PAYLOAD_SIZE（1452 字节）。

- 你可以使用 TCPSegment::length_in_sequence_space()方法来计算一个段所占用的序列号的总数。

2. `void ack_received( const WrappingInt32 ackno, const uint16_t window_size)`

- TCPSender 应该检查它的未完成的段集合，并删除任何现在已被完全确认的(确认号大于段中的所有序列号)。如果有了新的空间，TCPSender 应该再次填充窗口。

3. `void tick( const size_t ms_since_last_tick ):`

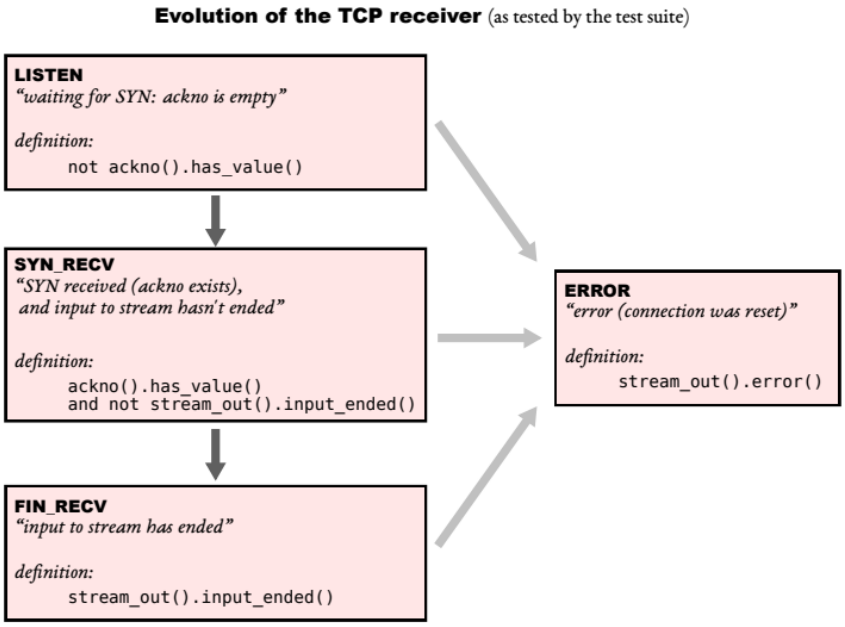
- 从上次调用该方法到现在已经经过了一定的毫秒数。发送方可能需要重传未完成的段。

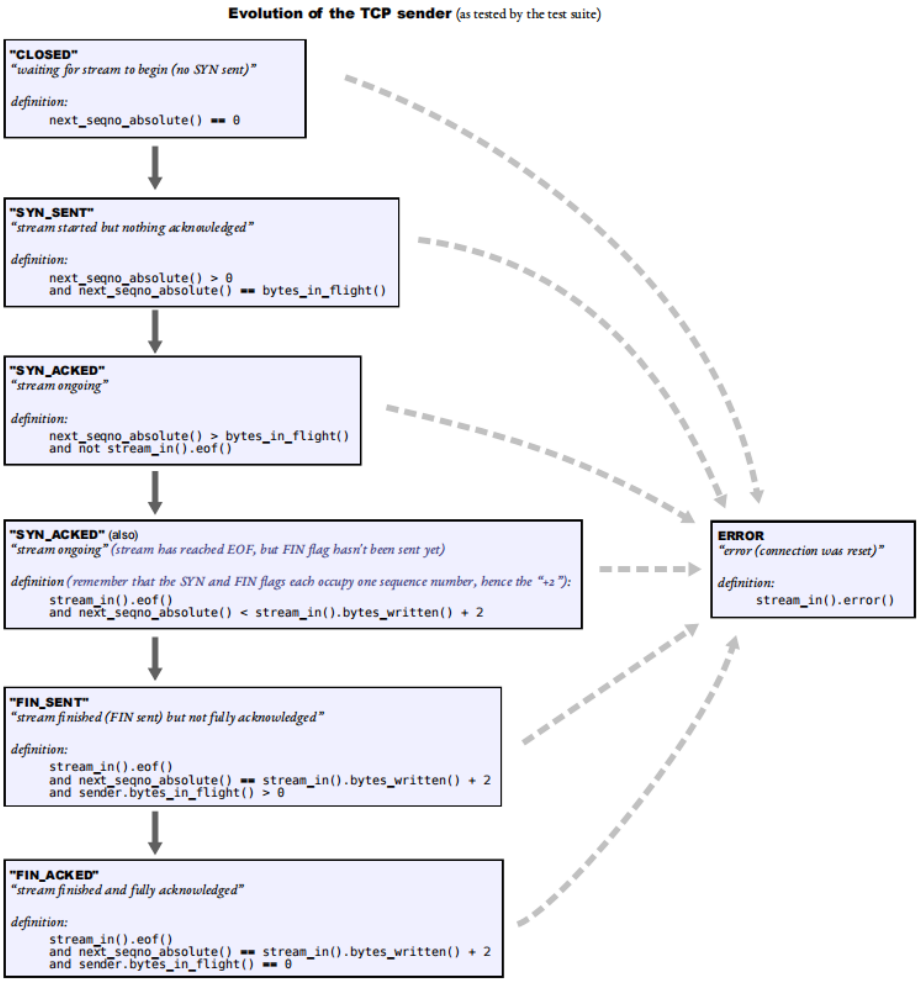
4. `void send_empty_segment():`

- TCPSender 应该生成并发送一个在序列空间中长度为零的 TCPSegment，并且序列号设置正确。如果想要发送一个空的 ACK 段，这是很有用的。

- 实现 TCPReceiver 和 TCPSender 后，运行 make check_lab2 命令以检查代码的正确性。

- 下面这两张图展示了 TCPReceiver 和 TCPSender 的测试流程，能够帮助你更好地理解：





- 温馨提示：当你在开发代码的时候，可能会遇到无法解决的问题，下面给出解决的办法。

- 运行 **cmake .. -DCMAKE_BUILD_TYPE=RelASan** 命令配置 build 目录，使编译器能够检测内存错误和未定义的行为并给你很好的诊断。

- 你还可以使用 valgrind 工具。

- 你也可以运行 **cmake .. -DCMAKE_BUILD_TYPE=Debug** 命令配置并使用 GNU 调试器（**gdb**）。

- 你可以运行 **make clean** 和 **cmake .. -DCMAKE_BUILD_TYPE=Release** 命令重置构建系统。

- 如果你不知道如何修复遇到的问你题，你可以运行 **rm -rf build** 命令删除 build 目录，创建一个新的 build 目录并重新运行 **cmake..**命令。

五、 实验数据记录和处理

以下实验记录均需结合屏幕截图（截取源代码或运行结果），进行文字标注（看完请删除本句）。

- 实现 WrappingInt32 的关键代码截图

```cpp
libsponge > wrapping_integers.cc > ...
#include "wrapping_integers.hh"

// Dummy implementation of a 32-bit wrapping integer

// For Lab 2, please replace with a real implementation that passes the
// automated checks run by `make check_lab2`.

template <typename... Targs>
void DUMMY_CODE(Targs &&... /* unused */) {}

using namespace std;

//! Transform an "absolute" 64-bit sequence number (zero-indexed) into a WrappingInt32
//! \param n The input absolute 64-bit sequence number
//! \param isn The initial sequence number
WrappingInt32 wrap(uint64_t n, WrappingInt32 isn) {
    return WrappingInt32{isn+static_cast<uint32_t>(n)};
}

//! Transform a WrappingInt32 into an "absolute" 64-bit sequence number (zero-indexed)
//! \param n The relative sequence number
//! \param isn The initial sequence number
//! \param checkpoint A recent absolute 64-bit sequence number
//! \returns the 64-bit sequence number that wraps to `n` and is closest to `checkpoint`
//!
//! \note Each of the two streams of the TCP connection has its own ISN. One stream
//! runs from the local TCPSender to the remote TCPReceiver and has one ISN,
//! and the other stream runs from the remote TCPSender to the local TCPReceiver and
//! has a different ISN.
uint64_t unwrap(WrappingInt32 n, WrappingInt32 isn, uint64_t checkpoint) {
    int32_t interval=n-wrap(checkpoint,isn);
    int64_t result=checkpoint+interval;
    if(result>=0)return result;
    else return result+(1ul<<32);
}
```

- 测试 Wrapping 的运行截图

```
cjwjiwang@cjwjiwang-virtual-machine:~/lab2/sponge/build$ ctest -R wrap
Test project /home/cjwjiwang/lab2/sponge/build
    Start 1: t_wrapping_ints_cmp
1/4 Test #1: t_wrapping_ints_cmp ..............   Passed    0.01 sec
    Start 2: t_wrapping_ints_unwrap
2/4 Test #2: t_wrapping_ints_unwrap ...........   Passed    0.01 sec
    Start 3: t_wrapping_ints_wrap
3/4 Test #3: t_wrapping_ints_wrap .............   Passed    0.01 sec
    Start 4: t_wrapping_ints_roundtrip
4/4 Test #4: t_wrapping_ints_roundtrip ........   Passed    0.18 sec

100% tests passed, 0 tests failed out of 4

Total Test time (real) =   0.22 sec
cjwjiwang@cjwjiwang-virtual-machine:~/lab2/sponge/build$
```

- 实现 TCPReceiver 的关键代码截图

  TCPReceiver.hh 中的私有变量

```
libsponge > tcp_receiver.hh > TCPReceiver > _isn
  1  #ifndef SPONGE_LIBSPONGE_TCP_RECEIVER_HH
 10
 11  //! \brief The "receiver" part of a TCP implementation.
 12
 13  //! Receives and reassembles segments into a ByteStream, and computes
 14  //! the acknowledgment number and window size to advertise back to the
 15  //! remote TCPSender.
 16  class TCPReceiver {
 17      //! Our data structure for re-assembling bytes.
 18      StreamReassembler _reassembler;
 19
 20      //! The maximum number of bytes we'll store.
 21      size_t _capacity;
 22
 23      //! The initial sequence number of the sender.
 24      std::optional<WrappingInt32> _isn;
 25      // bool _set_syn_flag;
 26
 27    public:
 28      //! \brief Construct a TCP receiver
 29      //!
```

TCPReceiver.cc:

```
libsponge > tcp_receiver.cc > window_size() const
  1  #include "tcp_receiver.hh"
  2
  3  // Dummy implementation of a TCP receiver
  4
  5  // For Lab 2, please replace with a real implementation that passes the
  6  // automated checks run by `make check_lab2`.
  7
  8  template <typename... Targs>
  9  void DUMMY_CODE(Targs &&... /* unused */) {}
 10
 11  using namespace std;
 12
 13  void TCPReceiver::segment_received(const TCPSegment &seg) {
 14      const auto &header = seg.header();
 15      if (!_isn.has_value()) {
 16          if (!header.syn) return; // SYN 之前的包都丢弃
 17          _isn = header.seqno; // 建立连接的 seqno 即是 ISN，也是 SYN 对应的 seqno
 18      }
 19      uint64_t checkpoint = _reassembler.stream_out().bytes_written();
 20      uint64_t abs_seq = unwrap(header.seqno, _isn.value(), checkpoint);
 21      uint64_t stream_index = abs_seq - 1 + (header.syn ? 1 : 0);
 22      _reassembler.push_substring(seg.payload().copy(), stream_index, header.fin);
 23  }
 24
 25  optional<WrappingInt32> TCPReceiver::ackno() const {
 26      if (!_isn.has_value()) return nullopt;
 27      uint64_t abs_seq = _reassembler.stream_out().bytes_written() + 1 + \
 28                         (_reassembler.stream_out().input_ended() ? 1 : 0);
 29      return wrap(abs_seq, _isn.value());
 30  }
 31
 32  size_t TCPReceiver::window_size() const {
 33      return _capacity - _reassembler.stream_out().buffer_size();
 34  }
```

- 实现 TCPSender 的关键代码截图

  TCPSender.hh:

libsponge > tcp_sender.hh > Timer > _time_count

```cpp
#ifndef SPONGE_LIBSPONGE_TCP_SENDER_HH
#include "byte_stream.hh"

#include "tcp_segment.hh"
#include "wrapping_integers.hh"

#include <cstdint>
#include <deque>
#include <functional>
#include <queue>
#include <utility>

//! \brief TCPSender 的计时器，最长定时时间 (ms) 不能超过 uint32
class Timer {
private:
    uint32_t _time_count = 0;
    uint32_t _time_out = 0;
    bool _is_running = false;
public:
    Timer() = default;
    Timer(const uint32_t time_out) : _time_out(time_out) {}
    // void start() { _is_running = true; }
    void stop() { _is_running = false; }
    void set_time_out(const uint32_t time_out) { _time_out = time_out; }
    uint32_t get_time_out() const { return _time_out; }
    void restart() { _is_running = true, _time_count = 0; }
    void tick(const size_t ms_since_last_tick) {
        if (_is_running)
            _time_count += ms_since_last_tick;
    }
    bool check_time_out() const { return _is_running && _time_count >= _time_out; }
    bool is_running() const { return _is_running; }
};
```

```cpp
class TCPSender {
  private:
    //! our initial sequence number, the number for our SYN.
    WrappingInt32 _isn;

    //! outbound queue of segments that the TCPSender wants sent
    std::queue<TCPSegment> _segments_out{};

    //! retransmission timer for the connection
    unsigned int _initial_retransmission_timeout;

    //! outgoing stream of bytes that have not yet been sent
    ByteStream _stream;

    //! the (absolute) sequence number for the next byte to be sent
    uint64_t _next_seqno{0};

    //! 重传定时器
    Timer _timer;

    //! 已经发出但还未收到 ACK 确认的 TCPSegment 队列
    std::queue<std::pair<uint64_t, TCPSegment> > _outstanding_seg{};

    //! 连续重传次数
    uint32_t _consecutive_retransmissions_count = 0;

    //! 已经发送出去但还未收到 ACK 确认的字节数
    size_t _bytes_in_flight = 0;

    //! 窗口大小，根据文档初始值应为 1
    uint16_t _window_size = 1;

    //! 是否发送带 SYN/FIN 的包
    bool _set_syn_flag = false, _set_fin_flag = false;
```

TCPSender.cc:

```cpp
void TCPSender::fill_window() {
    uint16_t window_size = max(_window_size, static_cast<uint16_t>(1));
    while (_bytes_in_flight < window_size) {
        TCPSegment seg;
        // 首先发 SYN 包，不含 payload（因为初始时 window_size 为 1）
        if (!_set_syn_flag) {
            seg.header().syn = true;
            _set_syn_flag = true;
        }

        // MAX_PAYLOAD_SIZE 只限制字符串长度并不包括 SYN 和 FIN，但是 window_size 包括 SYN 和 FIN
        auto payload_size = min(TCPConfig::MAX_PAYLOAD_SIZE, \
                                min(window_size - _bytes_in_flight - seg.header().syn, _stream.buffer_size()));
        auto payload = _stream.read(payload_size);
        seg.payload() = Buffer(std::move(payload));

        // 如果读到 EOF 了且 window_size 还有空位
        if (!_set_fin_flag && _stream.eof() && _bytes_in_flight + seg.length_in_sequence_space() < window_size) {
            seg.header().fin = true;
            _set_fin_flag = true;
        }

        // 空数据报就不发送了
        uint64_t length;
        if ((length = seg.length_in_sequence_space()) == 0) break;

        // 发送
        seg.header().seqno = next_seqno(); // next_seqno() 是 TCP seqno
        _segments_out.push(seg);

        // 如果定时器关闭，则启动定时器
        if (!_timer.is_running()) _timer.restart();

        // 保存备份，重发时可能会用
        _outstanding_seg.emplace(_next_seqno, std::move(seg));

        // 更新序列号和发出但未 ACK 的字节数
        _next_seqno += length; // _next_seqno 是 absolute seqno
        _bytes_in_flight += length;
    }
}

//! \param ackno The remote receiver's ackno (acknowledgment number)
//! \param window_size The remote receiver's advertised window size
void TCPSender::ack_received(const WrappingInt32 ackno, const uint16_t window_size) {
    auto abs_ackno = unwrap(ackno, _isn, next_seqno_absolute());
    if (abs_ackno > next_seqno_absolute()) return; // 传入的 ACK 是不可靠的，直接丢弃
    int is_successful = 0;

    // 处理已经收到的包（序列号空间要小于 ACK）
    while (!_outstanding_seg.empty()) {
        auto &[abs_seq, seg] = _outstanding_seg.front();
        if (abs_seq + seg.length_in_sequence_space() - 1 < abs_ackno) {
            is_successful = 1;
            _bytes_in_flight -= seg.length_in_sequence_space();
            _outstanding_seg.pop();
        } else {
            break;
        }
    }

    // 有成功 ACK 的包，则重置定时器，清零连续重传次数
    if (is_successful) {
        _consecutive_retransmissions_count = 0;
        _timer.set_time_out(_initial_retransmission_timeout);
        _timer.restart();
    }

    // 没有等待 ACK 的包了，则关闭定时器
    if (_bytes_in_flight == 0) {
        _timer.stop();
    }
```

```cpp
    // 更新 window_size，并尝试填满窗口
    _window_size = window_size;
    fill_window();
}

//! \param[in] ms_since_last_tick the number of milliseconds since the last call to this method
void TCPSender::tick(const size_t ms_since_last_tick) {
    _timer.tick(ms_since_last_tick);

    // 定时器超时（已经确保定时器已经打开），如果定时器关闭不会超时检查不会返回 true
    // 理论上不用检测 _outstanding_seg 非空，但为了鲁棒性就检测下吧
    if (_timer.check_time_out() && !_outstanding_seg.empty()) {
        // 重传最早的报文
        _segments_out.push(_outstanding_seg.front().second);

        // window_size 非 0 对应的操作
        if (_window_size > 0) {
            ++_consecutive_retransmissions_count;
            _timer.set_time_out(_timer.get_time_out() * 2);
        }

        // 重启定时器
        _timer.restart();
    }
}

unsigned int TCPSender::consecutive_retransmissions() const { return _consecutive_retransmissions_count; }

void TCPSender::send_empty_segment() {
    // 发送空数据报，可以用于仅仅 ACK
    TCPSegment seg;
    seg.header().seqno = next_seqno();
    _segments_out.emplace(std::move(seg));
}
```

- 运行 make check_lab2 命令的测试结果展示

```
28/34 Test #28: t_byte_stream_two_writes .........   Passed    0.00 sec
      Start 29: t_byte_stream_capacity
29/34 Test #29: t_byte_stream_capacity ...........   Passed    0.30 sec
      Start 30: t_byte_stream_many_writes
30/34 Test #30: t_byte_stream_many_writes ........   Passed    0.01 sec
      Start 31: t_webget
31/34 Test #31: t_webget .........................   Passed    1.17 sec
      Start 53: t_address_dt
32/34 Test #53: t_address_dt .....................   Passed    0.01 sec
      Start 54: t_parser_dt
33/34 Test #54: t_parser_dt ......................   Passed    0.00 sec
      Start 55: t_socket_dt
34/34 Test #55: t_socket_dt ......................   Passed    0.01 sec

100% tests passed, 0 tests failed out of 34

Total Test time (real) =   2.05 sec
[100%] Built target check_lab2
cjwjiwang@cjwjiwang-virtual-machine:~/lab2/sponge/build$
```

六、 实验数据记录和处理

- 通过代码，请描述 TCPSender 是如何发送出一个 segment 的？

  调用 fill_window 函数，根据当前的窗口大小和发送缓存区的数据情况，创建并发送新的 TCP 数据段。

```
uint16_t window_size = max(_window_size,
static_cast<uint16_t>(1));
while (_bytes_in_flight < window_size) {
    TCPSegment seg;
```

首先初始化并检查发送条件，最小窗口为一个字节。再判断已发送数据量是否小于窗口大小，若是，则进入循环准备发送数据。

```
if (!_set_syn_flag) {
    seg.header().syn = true;
    _set_syn_flag = true;
}
```

第一次发送时（即尚未发送过 SYN 包），SYN 标志被设置为 true。此时数据段仅含有 SYN 标志，没有有效载荷，并且只发送一次 SYN 包。

```
auto payload_size = min(TCPConfig::MAX_PAYLOAD_SIZE,
                        min(window_size - _bytes_in_flight -
seg.header().syn, _stream.buffer_size()));
auto payload = _stream.read(payload_size);
seg.payload() = Buffer(std::move(payload));
```

计算有效载荷大小，选择允许的最大值（MAX_PAYLOAD_SIZE）,剩余窗口空间（window_size-_bytes_in_flight）和当前缓存区可用数据量（_stream.buffer_size()），这三个的最小值。然后读取数据并设置为有效载荷。

```
if (!_set_fin_flag && _stream.eof() && _bytes_in_flight +
seg.length_in_sequence_space() < window_size) {
    seg.header().fin = true;
    _set_fin_flag = true;
}
```

结束连接时，如果发送端的缓冲区数据已经读取完毕，并且当前 TCP 数据段的大小小于窗口大小，则设置 FIN 标志。FIN 标志表示连接的结束，用于告诉接收方数据发送完毕。

```
uint64_t length;
if ((length = seg.length_in_sequence_space()) == 0) break;
```

过滤为空的数据包。

```
seg.header().seqno = next_seqno();
```

_segments_out.push(seg);

设置序列号并发送数据段，把 seg 放入待发送数据段队列。

if (!_timer.is_running()) _timer.restart();

如果定时器（用于监控超时重传的情况）没有启动，就启动。

_outstanding_seg.emplace(_next_seqno, std::move(seg));

保存未确认的数据段，以便于发生超时的时候重传。

_next_seqno += length;

_bytes_in_flight += length;

更新序列号和已经发送的字节数。

循环继续，知道发送的字节数达到窗口大小或者没有更多数据需要发送为止。

- 请用自己的理解描述一下 TCPSender 超时重传的整个流程。

  要发送的数据段先被放入_segments_out 队列中，准备发送，每个发送的段也会被存储在_outstanding_seg 队列中，这个队列用于追踪已经发送但还没有收到确认（ACK）的数据段。
  在每次调用 fill_window（）发送数据时，TCPSender 会检测定时器有无启动，没有启动的话就启动。
  接收方接收到数据段后，会发送一个带有确认号（ACK number）的 ACK 数据包回到发送方，告知接收方成功接收到的数据段的序列号。如果发送方收到了 ACK，它就会移除相应的已确认数据段，并根据接收到的 ACK 更新已发送字节数。
  如果定时器的超时时间到了，并且仍有未被确认的数据段（即_outstanding_seg 非空），则会进行重传。如果定时器超时，并且仍然有数据段没有收到确认（即_outstanding_seg 中有数据段），TCPSender 就会重传队列中最早发送的数据（_outstanding_seg.front().second）。每次重传时 TCPSender 会将超时的数据段重新放入_segments_out 队列进行发送。重传时，发送方会增加 consecutive_retransmissions_count，表示连续重传的次数：此外，每次重传时，定时器的超时时间会加倍（即指数回退），目的是避免网络拥塞过多的重传导致的性能下降。定时器的超时时间会逐渐变长，直到收到确认。
  每次重传时，定时器都会重新启动，当所有已发送数据段都已经被确认，定时器会停止。

- 请描述一下你为了重传未被确认的段建立的数据结构？为什么？

  为了实现重传未被确认的段，我使用了一个名为_outstanding_seg 的数据结构。这个数据结构是一个 std::queue，存储了所有已经发送但尚未

被确认（ACK）的 TCP 段。每个元素是一个 pair，包含 abs_seq:当前段的绝对序列号，和 seg:当前 TCP 段。
队列的使用保证了按顺序处理待重传的数据段，TCP 协议要求将最早发送的未确认段重传。
我还通过维护一个计数器_consecutive_retransmissions_count 来记录连续的重传次数。每当重传时，重传超时的时间会加倍（通过 set_time_out）来应对网络延迟或丢包等情况。

- 请思考为什么 TCPSender 实现中有一个发送空段的方法，能描述一下你是怎么理解的吗？

  TCP 协议传输过程中可能会有一些情况需要发送一个空的 ACK 包，对接收方发送的包进行确认。还可以用于快速确认已经接收到的数据段，避免不必要的重传。
  发送空段能够在不发送实际数据的情况下，维持 TCP 连接的健康状态，确保接收方正确处理 ACK 序列。

七、 实验数据记录和处理

在实验过程中，我遇到的最大困难时不理解一开始写好的 stream_reassembler.cc 文件，从而在 receiver 的调用中遇到困难。此外，一开始序列号的转换我也比较懵，通过阅读和学习才学会了序列号的运用。还有 TCPsender 中重传机制的实现，我只理解需要重传，但是没有设计合理的数据结构来帮助重传。