

合肥工业大学

《计算机网络系统实践》报告

设计题目：手写 TCP

学生姓名：余梓俊

学 号：2018211991

专业班级：计算机科学与技术 18-3 班

2021 年 3 月

目录

一、设计要求.....	4
二、开发环境与工具.....	4
三、设计原理.....	4
1.TUN 设备.....	5
2.TCPConnection 类.....	6
3.TCPSender 和 TCPReceiver 类.....	10
4.TCPSender、TCPReceiver 的内部.....	12
四、系统功能描述及软件模块划分.....	13
五、设计步骤.....	15
1.ByteString.....	15
2.StreamReassembler.....	16
3.WrappingInt32.....	21
4.TCPReceiver.....	23
5.TCPSender.....	25
6.TCPConnection.....	30
六、关键问题及其解决方法.....	32
七、设计结果.....	34
八、软件使用说明.....	36
九、参考资料.....	38
十、验收时间及验收情况.....	38
十一、设计体会.....	38

一、设计要求

自己实现一个 TCP，满足 RFC 文档标准，稳健地处理各种边缘情况，能够与真实世界的服务器建立 TCP 连接进行通讯。

二、开发环境与工具

语言：C++；

编译器：GCC9.3.0；

平台：Ubuntu 20.04(on WSL2)；

IDE：VSCode；

版本控制：Git。

三、设计原理

1. 首先我们希望利用 Linux 的 TUN 设备来收发我们的数据；

2. TUN 设备是一个三层设备，它模拟到了 IP 层，我们向 TUN 设备发送我们自己实现的 IP 数据报（这一部分涉及到的 File Descriptor、TUN/TAP 等底层代码由原课程实验提供）；

3. 在我们自己实现的 IP 数据包内，在 payload 中填入我们自己实现的 TCP 报文段；

4. 我们将自己实现 TCPConnection 类，类示例可以作为 TCP 连接的 peer：a. 具备同时收发的功能，可以处理三次握手和四次挥手过程（实例内部再实例化自己写的 TCPSender 和 TCPReceiver），b. 可以重组收到的有重叠的乱序报文段（内部实例化一个自己写的流重组器）；

5. 最后，我们将用自己写的 TCPConnection 实例，与真实世界的服务器建立 TCP 连接进行通讯（TCPConnection 将被封装在 TCPSpongeSocket 类中，用以模拟 UNIX 标准的 socket api 接口，且是多线程的，这一步封装也由原课程实验提供）；

6. 进行 profiling，我们的 TCPConnection 在本地的 benchmark 将被调优到 2G/s 以上。

下面对设计原理进行更详细的说明：

1. TUN 设备

TUN 设备是一种虚拟网络设备，通过此设备，程序可以方便地模拟网络行为。TUN 模拟的是一个三层设备，也就是说，通过它可以处理来自网络层的数据，更通俗一点的说，通过它，我们可以处理 IP 数据包。

TUN 的工作方式：

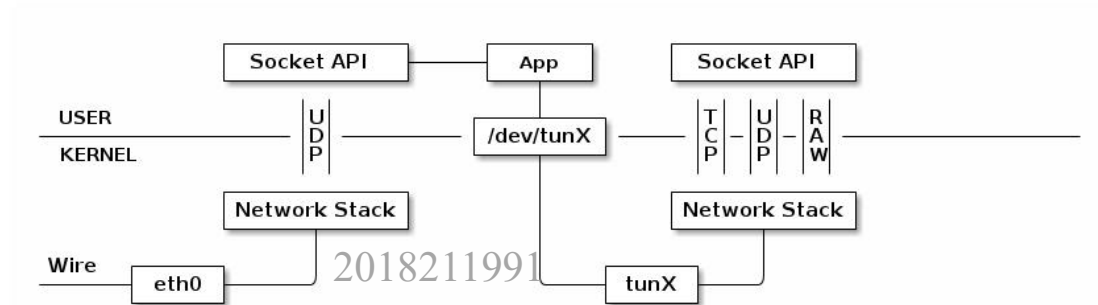


图 1 TUN 设备工作原理

普通的网卡是通过网线来收发数据包的话，而 TUN 设备比较特殊，它通过一个文件收发数据包。如上图所示，tunX 和 eth0 在逻辑上面是等价的，tunX 也代表了一个网络接口，虽然这个接口是系统通过软件所模拟出来的。

网卡接口 tunX 所代表的虚拟网卡通过文件 /dev/tunX 与我们的应用程序 (App) 相连，应用程序每次使用 write 之类的系统调用将数据写入该文件，这些数据会以网络层数据包的形式，通过该虚拟网卡，经由网络接口 tunX 传递给网络协议栈，同时该应用程序也可以通过 read 之类的系统调用，经由文件 /dev/tunX 读取到协议栈向 tunX 传递的所有数据包。

实现过程是自底向上的，先实现了字节流，再实现了字节流重组器，再实现 TCPSender、TCPReceiver、TCPConnection。但为了更清晰地描述整体架构与流程，接下来进行自顶向下的介绍。

a. 接收报文。当它的段接收方法被调用时，TCPConnection 从网络层接收 TCPSegments，TCPConnection 会查看该段，并：

- 如果设置了 rst (reset) 标志，会将入站和出站流都设置为错误状态，并永久终止连接。否则
- 将该段提供给 TCPReceiver，以便它可以检查它关心的传入段上的字段：seqno、syn、payload 和 fin。
- 如果设置了确认标志，则告知 TCPSender 它关心的传入段的字段：ackno 和窗口大小。
- 如果传入的数据段占用了任何序列号，TCPConnection 会确保至少有一个数据段作为回复发送，以反映 ackno 和窗口大小的更新。

b. 发送报文。

- 每当 TCPSender 将一个段推到它的输出队列中，并设置了它负责输出段的字段：(seqno、syn、有效载荷和 fin)。
- 在发送数据段之前，TCPConnection 将向 TCPReceiver 询问它在传出数据段上负责的字段：确认和窗口大小。如果有确认，它将设置确认标志和 TCPSegment 中的字段。

c. 处理时间的流逝。TCPConnection 有一个 tick 方法，操作系统将定期调用该方法。此时，TCPConnection 需要：

- 告诉 TCPSender 关于时间的流逝。
- 如果连续重传的次数超过上限 TCPConfig::MAX_RETX_ATTEMPTS，则中止连接，并向对等方发送重置段(设置了 rst 标志的空段)。

因此，TCPSegment 的整体结构如下所示，TCPSender 和 TCPReceiver 各自负责填写的字段以不同的颜色显示：

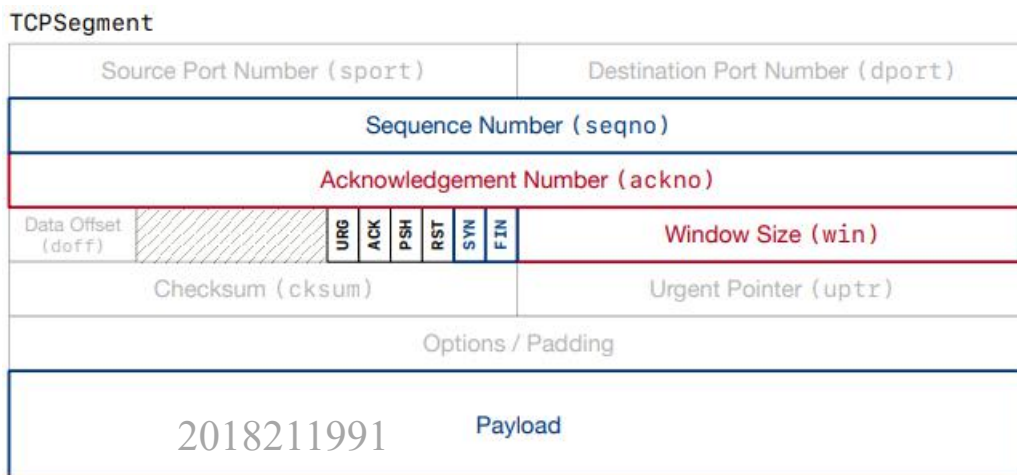


图 4 TCP 报文段格式

在实现过程中，需要额外关注收到报文段时 TCPSender 和 TCPConnection 的逻辑的不同之处。这些细节来源于

1. 自先前自底向上的实现过程中：receiver 只关心收到数据和数据有关的标识；sender 只关心收到的 ackno 和 win，不处理也不知道收到的数据和其他信息，在 `_stream_in()` 没有数据时可能不会做任何动作，而在此处，可能还需要发一个空的 ACK 报文段（也就是先前自底向上的实现需要一点额外的逻辑完善）

2. 连接建立和释放过程中的各种特殊情况

- a. 发完 SYN 后马上收到 RST
- b. 发完 SYN 后马上收到 FIN
- c. Simultaneous open
- d. Simultaneous shutdown
- ...

整个连接过程的自动机状态，以及 Simultaneous open 和 Simultaneous shutdown 的处理逻辑分别如下面三个图所示：

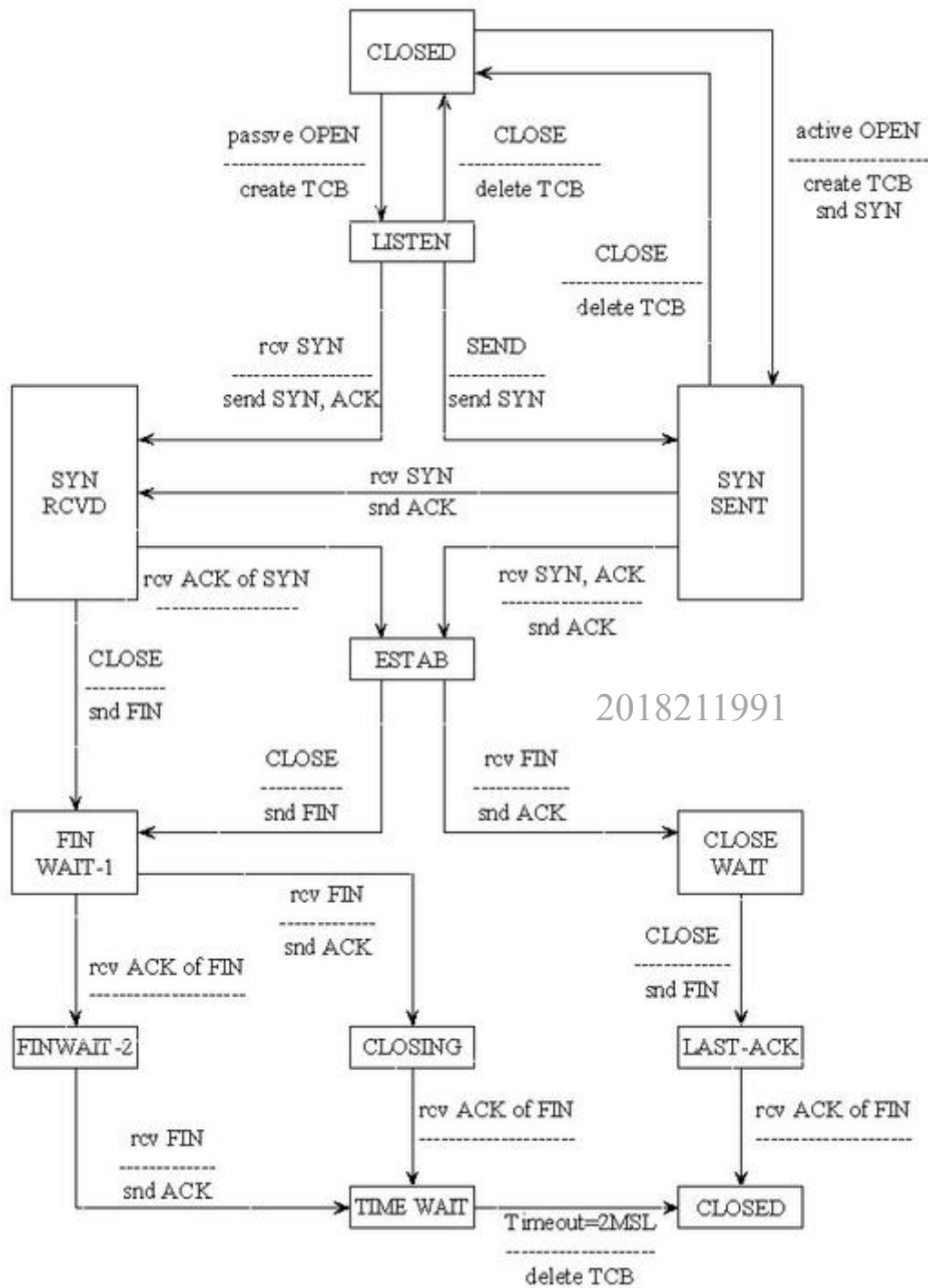


图5 FSM of TCP

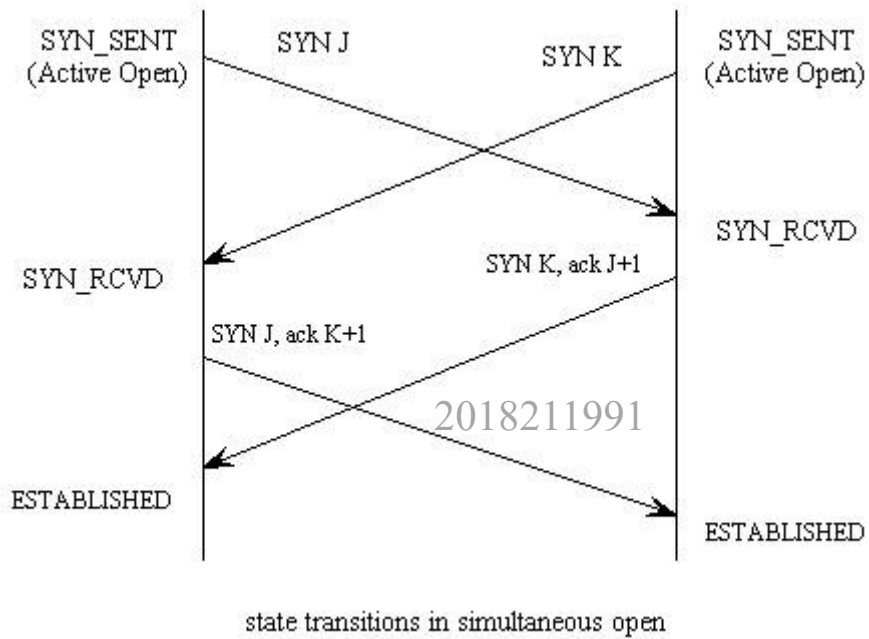


图 6 Simultaneous Open

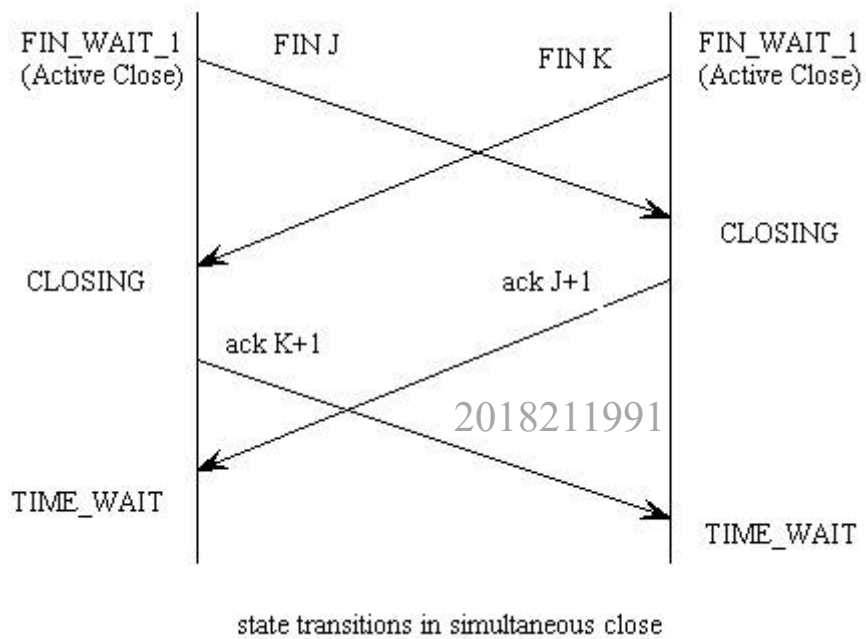


图 7 Simultaneous Close

3. TCPSender 和 TCPReceiver 类

上面简单说明了 TCPConnection 如何收发数据，这里给出更详细的 Sender

和 Receiver 的作用：

- 收到报文段时
 - 通知 `_receiver`：根据报文段的 `seqno`、`SYN`、`FIN` 和 `payload`，以及当前状态，更新 `ackno`；收集数据
 - 通知 `_sender`：根据报文段的 `ackno` 以及当前状态，更新 `next_seqno`；更新 `window_size`
- 发送报文段时
 - `_sender` 负责填充 `payload`、`seqno`、`SYN`、`FIN`，注意有可能既没有 `payload` 也没有 `S`、`F` 标识（empty segment）（再次说明，这和上一步自底向上实现的 `_sender` 的 `ack_received()` 逻辑不同）
 - `_receiver` 负责填充 `ackno`、`window size`

TCPReceiver 和 TCPSender 的状态转换图分别如下：

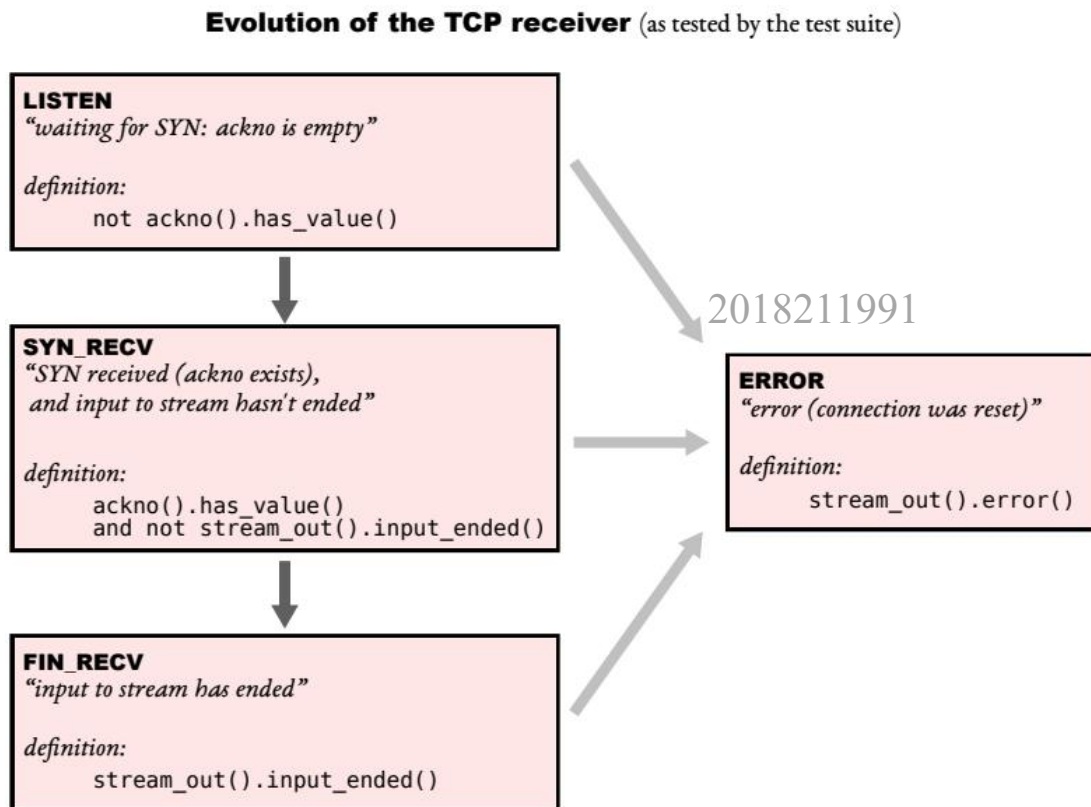


图 8 FSM of TCPReceiver

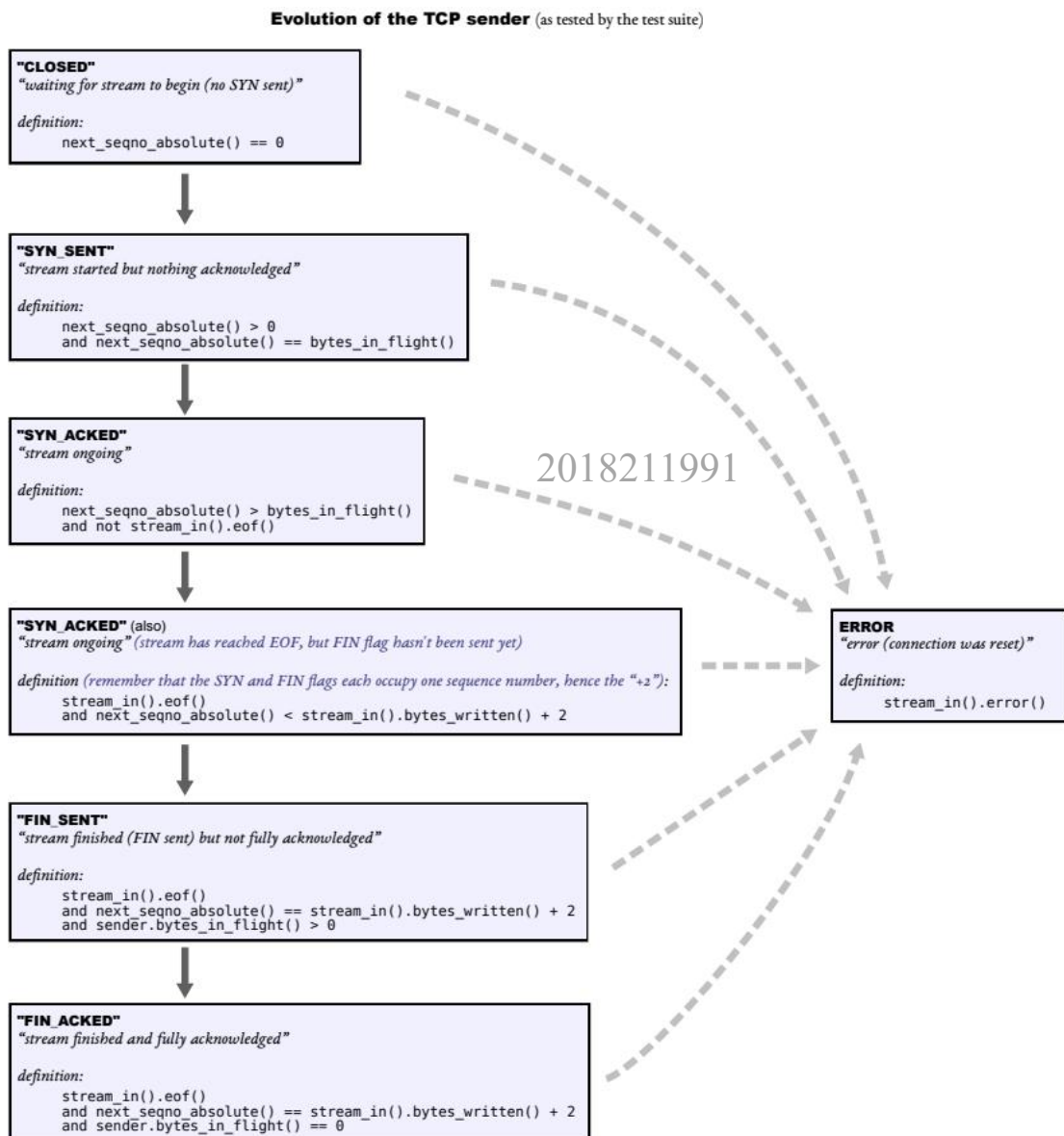


图9 FSM of TCPSender

4. TCPSender、TCPReceiver 的内部

首先，在图2中可以看到，二者内部都有一个我们自己写的 ByteStream 实例，Receiver 内部还需要在 ByteStream 外面套一个 StreamReassembler 实例。这两个类的 index 我们都将用 64 位无符号整数来存储，这足够大，能传输任何大小的数据，但是注意，TCP 报文段的 seqno 和 ackno，都为 32 位，因此我们还需要实现一个 WrappingInt32 类，以实现 64 位的 stream index 和 32 位的 seqno 之间的转换（再次说明，我们的自底向上实现先实现了字节流和重组器，然后实

现 WrappingInt32，利用它来正确将字节流封装在 TCPSender 和 TCPReceiver 内)。

这三个类 (ByteStream, StreamReassembler 和 WrappingInt32) 的设计和实现思路将在后面讲解。

四、系统功能描述及软件模块划分

libsponge 文件夹是我们的主要工作：

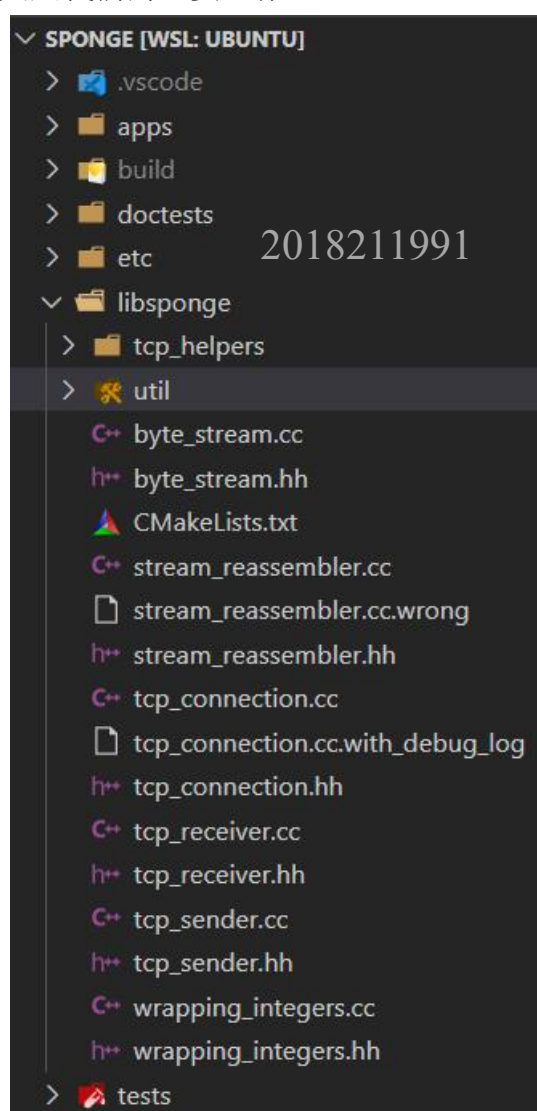


图 10 项目结构

其中 util 文件夹是最底层部分的代码，包括控制 TUN 设备的代码，重新封

装的 file_descriptor、socket、address 等等；

tcp_helpers 文件夹中包括实现的 ipv4_header, ipv4_datagrams, tcp_header, tcp_segment, tcp_state 等等（注意 tcp_state 中虽然定义了状态机中的所有状态，但我们实现连接的各种细节逻辑时，并不把状态机用代码实现，那样代码将会非常重，而是直接用一系列严密的条件语句实现所有状态的转换）。

下面是两个主要的类继承关系：

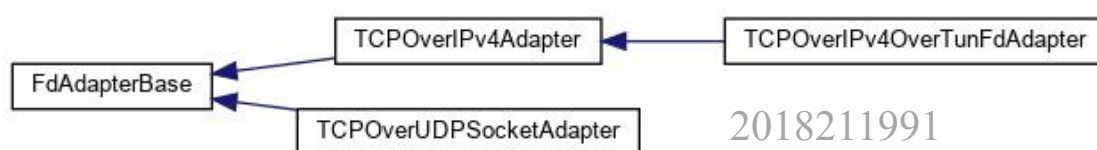


图 11 类继承关系-1

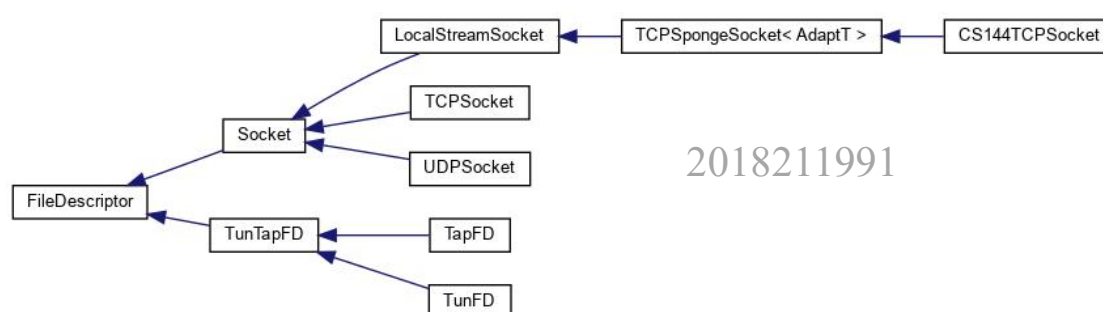


图 12 类继承关系-2

其中 FdAdapterBase 是一个基础的 file descriptor 适配器，TCPOverIPv4Adapter 是一个转换器，负责 TCP 报文段序列化成 IPv4 数据包，TCPOverIPv4OverTunFdAdapter 是一个更具体化的 file descriptor 适配器，负责在 TUN 设备上读写 IPv4 数据包。如设计原理部分第二条所述，该部分由原课程实验提供。

TCPSpongeSocket<AdaptT> 是一个类模板，它将实例化一个 TCPOverIPv4SpongeSocket 类，该类将封装我们的 TCPConnection，来模拟 UNIX 标准的 socket api 接口，且是多线程的，最后 CS144TCPSocket 是一个辅助类，它将使 TCPOverIPv4SpongeSocket 表现得更像普通的 Linux 内核的 TCPSocket。

如设计原理部分第五条所述，该部分也由原课程实验提供。

五、设计步骤

前文设计原理部分，考虑到叙述的清晰程度，采用了自顶向下的介绍，接下来的设计步骤，将重新采用自底向上的讲解，这与我真正实现的顺序相同。

1. ByteStream

正如前文所述，ByteStream 包括后面的 StreamReassembler 都将采用 64 位的 index 和其他各种辅助信息，以提供足够大的向上面的应用层传输数据的能力。

byte_stream.cc:

```
ByteStream::ByteStream(const size_t capacity) : _capacity(capacity) {}

size_t ByteStream::write(const string &data) {
    size_t write_count = 0;
    for (const char c : data) {
        // not very efficient to do conditional in loop
        if (_capacity - _buffer_size <= 0)
            break;
        else {
            _stream.push_back(c);
            ++_buffer_size;
            ++_bytes_written;
            ++write_count;
        }
    }

    return write_count;
}

///!!! \param[in] len bytes will be copied from the output side of the buffer
string ByteStream::peek_output(const size_t len) const {
    const size_t peek_length = len > _buffer_size ? _buffer_size : len;
    list<char>::const_iterator it = _stream.begin();
    advance(it, peek_length);
}
```

```

        return string(_stream.begin(), it);
    }

    ///! \param[in] len bytes will be removed from the output side of the buffer
    void ByteStream::pop_output(const size_t len) {
        size_t pop_length = len > _buffer_size ? _buffer_size : len;
        _bytes_read += pop_length;
        _buffer_size -= pop_length;
        while (pop_length--)
            _stream.pop_front();
    }

    ///! Read (i.e., copy and then pop) the next "len" bytes of the stream
    ///! \param[in] len bytes will be popped and returned
    ///! \returns a string
    std::string ByteStream::read(const size_t len) {
        const string result = peek_output(len);
        pop_output(len);
        return result;
    }

```

2. StreamReassembler

TCP 接受方接收到乱序且可能重叠的报文段，StreamReassembler 需要将收到的报文段按情况送入 ByteStream，或丢弃，或暂存（在合适的时候重组送入 ByteStream）。

注意点：

1. 报文段包含索引、长度、内容，labl 的索引从 0 开始增长，不会溢出绕回。
2. 任何报文段，包括新收到的和暂存的，只要可以，就应该立刻送入 ByteStream（可能需要手动重组和去重叠）。
3. 容量的限制如下图所示。

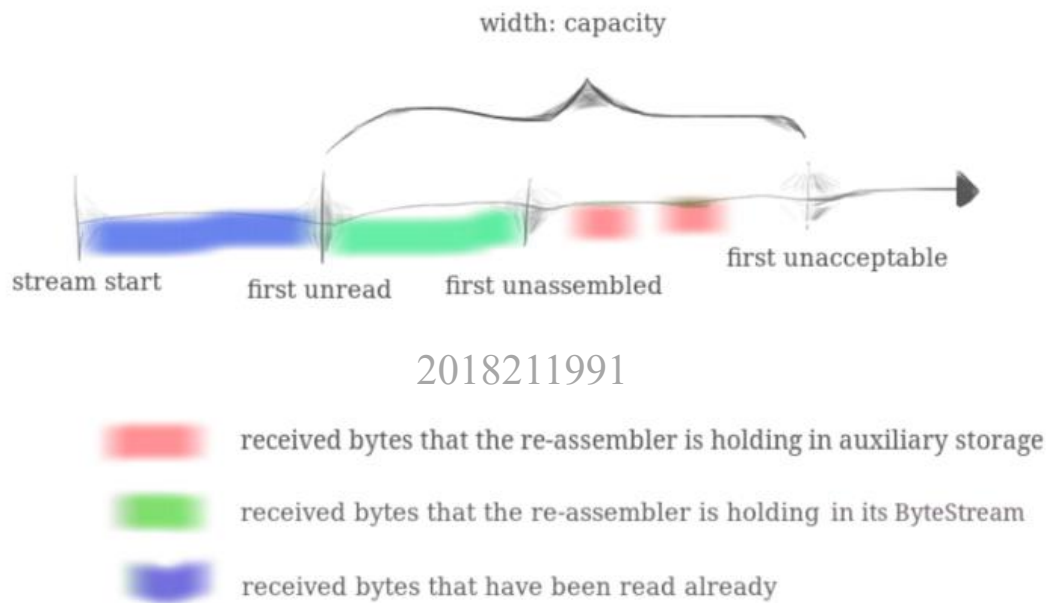


图 13 重组器

stream_reassembler.hh:

```
class StreamReassembler {
private:

    ByteStream _output; ///< The reassembled in-order byte stream
    size_t _capacity; ///< The maximum number of bytes
    size_t _first_unread = 0;
    size_t _first_unassembled = 0;
    size_t _first_unacceptable;
    bool _eof = false;
    struct seg {
        size_t index;
        size_t length;
        std::string data;
        bool operator<(const seg t) const { return index < t.index; }
    };
    std::set<seg> _stored_segs = {};

    void _add_new_seg(seg &new_seg, const bool eof);
    void _handle_overlap(seg &new_seg);
    void _stitch_output();
    void _stitch_one_seg(const seg &new_seg);
    void _merge_seg(seg &new_seg, const seg &other);

public:
```

实现思路如下：

- 用 `set` 来暂存报文段，按照报文段的 `index` 大小对比重载 `<` 运算符。
- 用 `_eof` 来保存是否已经收到过有 `EOF` 标识的段。
- 收到新段时，通过对比新段的 `index`、`length` 和 `_first_unacceptable`，`_first_unassembled` 对新段进行必要的剪切，然后处理重叠（调用 `_handle_overlap`）。
- 处理重叠的逻辑：遍历每个暂存段，如果与新段产生重叠，合并暂存段与新段（调用 `_merge_seg`，总是往新段上合并，合并后删除重叠的暂存段）。
- 合并段的方法：分类讨论，两个段总共会有四种不同的重叠情况，分别处理。
- 处理完重叠后，调用 `_stitch_output`：遍历所有暂存段，将可合并的暂存段接入 `ByteStream` 中。
- 最后，当 `_eof` 为真且 `unassembled_bytes` 为 0 时，调用 `ByteStream` 的 `end_input`。

`stream_reassembler.cc`:

```
StreamReassembler::StreamReassembler(const size_t capacity)
    : _output(capacity), _capacity(capacity), _first_unacceptable(capacity) {}

void StreamReassembler::_add_new_seg(seg &new_seg, const bool eof) {
    // check capacity limit, if unmeet limit, return
    // cut the bytes in NEW_SEG that will overflow the _CAPACITY
    // note that the EOF should also be cut
    // cut the bytes in NEW_SEG that are already in _OUTPUT
    // _HANDLE_OVERLAP()
    // update _EOF
    if (new_seg.index >= _first_unacceptable)
        return;
    bool eof_of_this_seg = eof;
    if (int overflow_bytes = new_seg.index + new_seg.length -
        _first_unacceptable; overflow_bytes > 0) {
        int new_length = new_seg.length - overflow_bytes;
        if (new_length <= 0)
            return;
        eof_of_this_seg = false;
    }
```

```

    new_seg.length = new_length;
    new_seg.data = new_seg.data.substr(0, new_seg.length);
}
if (new_seg.index < _first_unassembled) {
    int new_length = new_seg.length - (_first_unassembled - new_seg.index);
    if (new_length <= 0)
        return;
    new_seg.length = new_length;
    new_seg.data = new_seg.data.substr(_first_unassembled - new_seg.index,
new_seg.length);
    new_seg.index = _first_unassembled;
}
_handle_overlap(new_seg);
// if EOF was received before, it should remain valid
_eof = _eof || eof_of_this_seg;
}

```

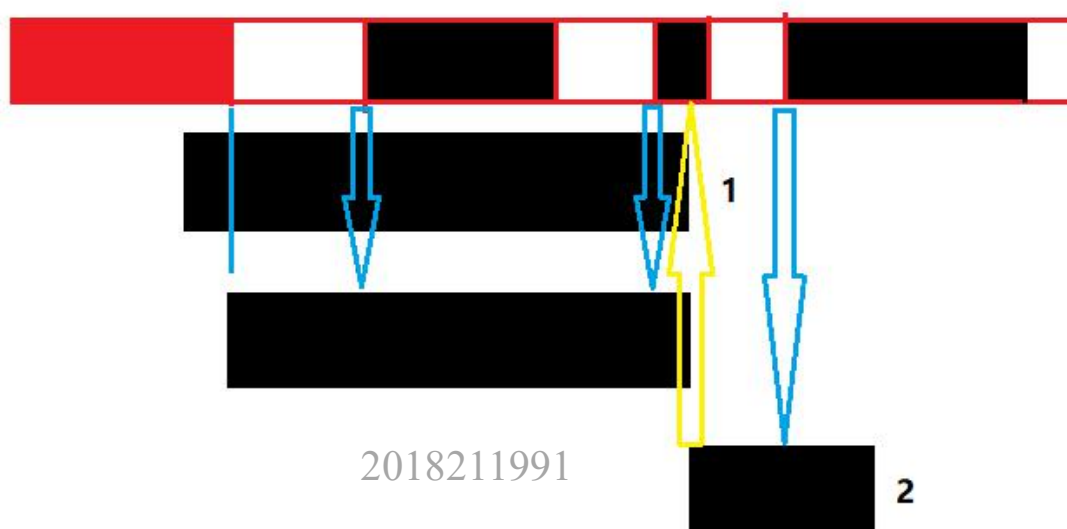


图 14 处理新到的报文段

```

void StreamReassembler::_handle_overlap(seg &new_seg) {
    for (auto it = _stored_segs.begin(); it != _stored_segs.end(); ) {
        auto next_it = ++it;
        --it;
        if ((new_seg.index >= it->index && new_seg.index < it->index + it->length)
||
            (it->index >= new_seg.index && it->index < new_seg.index +
new_seg.length)) {

```

```

        _merge_seg(new_seg, *it);
        _stored_segs.erase(it);
    }
    it = next_it;
}
_stored_segs.insert(new_seg);
}

void StreamReassembler::_merge_seg(seg &new_seg, const seg &other) {
    size_t n_index = new_seg.index;
    size_t n_end = new_seg.index + new_seg.length;
    size_t o_index = other.index;
    size_t o_end = other.index + other.length;
    string new_data;
    if (n_index <= o_index && n_end <= o_end) {
        new_data = new_seg.data + other.data.substr(n_end - o_index, n_end -
o_end);
    } else if (n_index <= o_index && n_end >= o_end) {
        new_data = new_seg.data;
    } else if (n_index >= o_index && n_end <= o_end) {
        new_data =
            other.data.substr(0, n_index - o_index) + new_seg.data +
other.data.substr(n_end - o_index, n_end - o_end);
    } else /* if (n_index >= o_index && n_end <= o_end) */ {
        new_data = other.data.substr(0, n_index - o_index) + new_seg.data;
    }
    new_seg.index = n_index < o_index ? n_index : o_index;
    new_seg.length = (n_end > o_end ? n_end : o_end) - new_seg.index;
    new_seg.data = new_data;
}
}

```

如图 14，红色部分为已经存入 ByteStream 的数据（但还未被 read() 读走），后面的黑色的三段是暂存的碎片报文。

若分别收到两个报文段 1 和 2。首先要裁剪，例如 1，将已经送入 ByteStream 却还在新报文中出现的部分裁剪掉（还要裁剪掉超出总容量范围的部分）。然后，遍历每个暂存的报文段，如果和新报文 overlap，则调用 _merge_seg() 函数。判断 overlap 的方法可以通过看这样的两个点：

1. 当前遍历的报文段的 seqno 在新报文段中间，即图中向下指的蓝色箭头。
2. 新报文段的 seqno 在当前遍历的报文段的中间，即图中向上的黄色箭头。

`_merge_seg()` 过程中，总是以新报文段的内容覆盖暂存报文的重叠部分。
遍历完后，更新暂存区。

然后，调用 `stitch_output()`，把更新后的暂存区中满足要求的报文段从暂存区送到 `ByteStream` 中。

```
void StreamReassembler::_stitch_output() {
    // _FIRST_UNASSEMBLED is the expected next index_FIRST_UNASSEMBLED
    // compare _STORED_SEGS.begin()->index with
    // if equals, then _STITCH_ONE_SEG() and erase this seg from set
    // continue compare until not equal or empty
    while (!_stored_segs.empty() && _stored_segs.begin()->index ==
_first_unassembled) {
        _stitch_one_seg(*_stored_segs.begin());
        _stored_segs.erase(_stored_segs.begin());
    }
}

void StreamReassembler::_stitch_one_seg(const seg &new_seg) {
    // write string of NEW_SEG into _OUTPUT
    // update _FIRST_UNASSEMBLED
    _output.write(new_seg.data);
    _first_unassembled += new_seg.length;
    // both way of updating _FIRST_UNASSEMBLED is ok
    // _first_unassembled = _output.bytes_written();
}

size_t StreamReassembler::unassembled_bytes() const {
    size_t unassembled_bytes = 0;
    for (auto it = _stored_segs.begin(); it != _stored_segs.end(); ++it)
        unassembled_bytes += it->length;
    return unassembled_bytes;
}

bool StreamReassembler::empty() const { return unassembled_bytes() == 0; }
```

3. WrappingInt32

前文介绍的 64 位的 `stream index` 和 32 位的 `segment seqno` 之间的转换关

系具体如下：

<i>element</i>	SYN	<i>c</i>	<i>a</i>	<i>t</i>	FIN
seqno	$2^{32} - 2$	$2^{32} - 1$	0	1	2
absolute seqno	0	1	2	3	4
stream index		0	1	2	

2018211991

The figure shows the three different types of indexing involved in TCP:

Sequence Numbers	Absolute Sequence Numbers	Stream Indices
<ul style="list-style-type: none">• Start at the ISN• Include SYN/FIN• 32 bits, wrapping• “seqno”	<ul style="list-style-type: none">• Start at 0• Include SYN/FIN• 64 bits, non-wrapping• “absolute seqno”	<ul style="list-style-type: none">• Start at 0• Omit SYN/FIN• 64 bits, non-wrapping• “stream index”

图 15 序号转换

实现思路：

首先看把 64 位的 index 转换为 WrappingInt32 的 wrap 函数。根据上面的表格，显然，只需要利用加法重载，把 ISN 往前走 $n(=index)$ 步就可以了。

然后看 unwrap 函数，起作用是把新接收到的报文段的 seqno (WrappingInt32)转换为 64 位的 index。seqno 转 index 的结果显然不唯一，我们想要的是与上一次收到的报文段的 index (checkpoint) 最接近的那个转换结果。于是，可以先用刚写好的 wrap 函数把 checkpoint 变为 WrappingInt32，然后利用第一个减法重载，找出这个转换后的 seqno 最少需要走几步可以到新报文的 seqno，然后把这个步数加到 checkpoint 上。注意这里有一个特殊情况（见代码注释），因为我们有可能是往数轴的反方向走的，可能走完之后 res 的值是个负数，这时候需要在加上一个 2^{32} 。

wrapping_integers.cc:

```
WrappingInt32 wrap(uint64_t n, WrappingInt32 isn) { return isn +
static_cast<uint32_t>(n); }

uint64_t unwrap(WrappingInt32 n, WrappingInt32 isn, uint64_t checkpoint) {
    // STEP ranges from -UINT32_MAX/2 to UINT32_MAX/2
    // in most cases, just adding STEP to CHECKPOINT will get the absolute seq
    // but if after adding, the absolute seq is negative, it should add another
    (1UL << 32)
    // (this means the checkpoint is near 0 so the new seq should always go bigger
```

```
// eg. test/unwrap.cc line 25)
int32_t steps = n - wrap(checkpoint, isn);
int64_t res = checkpoint + steps;
return res >= 0 ? checkpoint + steps : res + (1UL << 32);
}
```

4. TCPReceiver

这里重点关注 TCPReceiver 中三个函数的实现：

a. segment_received()

这是最主要的方法。每次从对方接收到新的段时，都会调用 TCPReceiver::segment_received()。

该方法需要：

- 如有必要，设置 ISN。设置了同步标志的第一个到达段的序列号是 ISN。为了在 32 位的 seqno/acknow 和 64 位的 index 之间进行转换，ISN 需要记录下来。
- 将任何数据或流结束标记推送到流重组器。如果 FIN 标志被设置在 TCPSegment 的报头中，这意味着有效载荷的最后一个字节是整个流的最后一个字节。

b. ackno()

返回一个 optional<WrappingInt32>，包含接收方不知道的第一个字节的序列号，即窗口的左边缘：接收者感兴趣接收的第一个字节。如果尚未设置 ISN，则返回一个空的 optional。

c. window_size()

等价于“first unassembled”和“first unacceptable”之间的距离。

TCPReceiver 要处理的特殊情况：

- SYN with DATA
- SYN with DATA with FIN
- SYN + FIN
- SYN + DATA with FIN + DATA
- ...

```

void TCPReceiver::segment_received(const TCPSegment &seg) {
    TCPHeader header = seg.header();
    if (header.syn && !_syn)
        return;
    if (header.syn) {
        _syn = true;
        _isn = header.seqno.raw_value();
    }
    // note that fin flag seg can carry payload
    if (_syn && header.fin)
        _fin = true;
    size_t absolute_seqno = unwrap(header.seqno, WrappingInt32(_isn),
    _checkpoint);
    _reassembler.push_substring(seg.payload().copy(), header.syn ? 0 :
absolute_seqno - 1, header.fin);
    _checkpoint = absolute_seqno;
}

optional<WrappingInt32> TCPReceiver::ackno() const {
    // Return the corresponding 32bit seqno of
    _reassembler.first_unassembled().
    // Translate from Stream Index to Absolute Sequence Number is simple, just
    add 1.
    // If outgoing ack is corresponding to FIN
    // (meaning FIN is received and all segs are assembled),
    // add another 1.
    size_t shift = 1;
    if (_fin && _reassembler.unassembled_bytes() == 0)
        shift = 2;
    if (_syn)
        return wrap(_reassembler.first_unassembled() + shift,
WrappingInt32(_isn));
    return {};
}

```



```

size_t TCPReceiver::window_size() const {
    // equal to first_unacceptable - first_unassembled
    return _capacity - stream_out().buffer_size();
}

```

5. TCPSender

TCPSender 的总体思路：

1. 我们要不需要实现选择重传，而是类似回退 N，需要存储已发送并且未被确认的段，进行累计确认，超时只要重传这些段中最早的那一个即可。

2. Sender 需要维护计时器：

- 发送新报文段时若计时器未打开，开启
- `ack_received()` 中，如果有报文段被正确地确认，重置计时器和 RTO，如果所有报文段均被确认 (`bytes in flight == 0`)，关闭计时器
- `tick()` 中，若计时器为关闭状态，直接返回，否则累加计时然后处理超时

`tcp_sender.hh`:

```

class TCPSender {
private:
    bool _syn_sent = false;
    bool _fin_sent = false;
    uint64_t _bytes_in_flight = 0;
    uint16_t _receiver_window_size = 0;
    uint16_t _receiver_free_space = 0;
    uint16_t _consecutive_retransmissions = 0;
    unsigned int _rto = 0;
    unsigned int _time_elapsed = 0;
    bool _timer_running = false;
    std::queue<TCPSegment> _segments_outstanding{};

    bool _ack_valid(uint64_t abs_ackno);
    void _send_segment(TCPSegment &seg);

```

```
};
```

- `send_segment(TCPSegment &seg)` 只在 `fill_window()` 中被调用, 重传只需要 `_segments_out.push(_segments_outstanding.front())`
- `_receiver_window_size` 保存收到有效 (有效的含义见上面 `ack_valid()`) 确认报文段时, 报文段携带的接收方窗口大小
- `_receiver_free_space` 是在 `_receiver_window_size` 的基础上, 再减去已发送的报文段可能占用的空间 (`_bytes_in_flight`)

`fill_window()` 实现:

- 如果 SYN 未发送, 发送然后返回
- 如果 SYN 未被应答, 返回
- 如果 FIN 已经发送, 返回
- 如果 `_stream` 暂时没有内容但并没有 EOF, 返回
- 如果 `_receiver_window_size` 不为 0
 1. 当 `receiver_free_space` 不为 0, 尽可能地填充 payload
 2. 如果 `_stream` 已经 EOF, 且 `_receiver_free_space` 仍不为 0, 填上 FIN (fin 也会占用 `_receiver_free_space`)
 3. 如果 `_receiver_free_space` 还不为 0, 且 `_stream` 还有内容, 回到步骤 1 继续填充
- 如果 `_receiver_window_size` 为 0, 则需要发送零窗口探测报文
 - 如果 `_receiver_free_space` 为 0
 - 如果 `_stream` 已经 EOF, 发送仅携带 FIN 的报文
 - 如果 `_stream` 还有内容, 发送仅携带一位数据的报文
 - 之所以还需要判断 `_receiver_free_space` 为 0, 是因为这些报文段在此处应该只发送一次, 后续的重传由 `tick()` 函数控制, 而当发送了零窗口报文段后 `_receiver_free_space` 的值就会从原来的与 `_receiver_window_size` 相等的 0 变成 -1

`ack_received()` 实现:

代码比较直白，见下方代码，注意进行累计确认之后，如果还有未被确认的报文段，`_receiver_free_space` 的值应为：收到的确认号绝对值 + 窗口大小 - 首个未确认报文的序号绝对值 - 未确认报文段的长度总和。

`tick()` 实现：

注意，窗口大小为 0 时不需要增加 `RTO`。但是发送 `SYN` 时，窗口为初始值也为 0，而 `SYN` 超时是需要增加 `RTO` 的。

`tcp_sender.cc`：

```
TCPSender::TCPSender(const size_t capacity, const uint16_t retx_timeout, const
std::optional<WrappingInt32> fixed_isn)
    : _isn(fixed_isn.value_or(WrappingInt32{random_device()}()))
    , _initial_retransmission_timeout{retx_timeout}
    , _stream(capacity)
    , _rto{retx_timeout} {}

void TCPSender::fill_window() {
    if (!_syn_sent) {
        _syn_sent = true;
        TCPSegment seg;
        seg.header().syn = true;
        send_segment(seg);
        return;
    }
    if (!_segments_outstanding.empty() &&
        _segments_outstanding.front().header().syn)
        return;
    if (!_stream.buffer_size() && !_stream.eof())
        return;
    if (_fin_sent)
        return;

    if (_receiver_window_size) {
        while (_receiver_free_space) {
            TCPSegment seg;
            size_t payload_size = min({_stream.buffer_size(),
static_cast<size_t>(_receiver_free_space),
static_cast<size_t>(TCPConfig::MAX_PAYLOAD_SIZE)});
```

```

        seg.payload() = _stream.read(payload_size);
        if (_stream.eof() && static_cast<size_t>(_receiver_free_space) >
payload_size) {
            seg.header().fin = true;
            _fin_sent = true;
        }
        send_segment(seg);
        if (_stream.buffer_empty())
            break;
    }
} else if (_receiver_free_space == 0) {
    // The zero-window-detect-segment should only be sent once
(retransmission excute by tick function).
    // Before it is sent, _receiver_free_space is zero. Then it will be -1.
    TCPSegment seg;
    if (_stream.eof()) {
        seg.header().fin = true;
        _fin_sent = true;
        send_segment(seg);
    } else if (!_stream.buffer_empty()) {
        seg.payload() = _stream.read(1);
        send_segment(seg);
    }
}
}
}

```

fill_window() 对应的流程图:

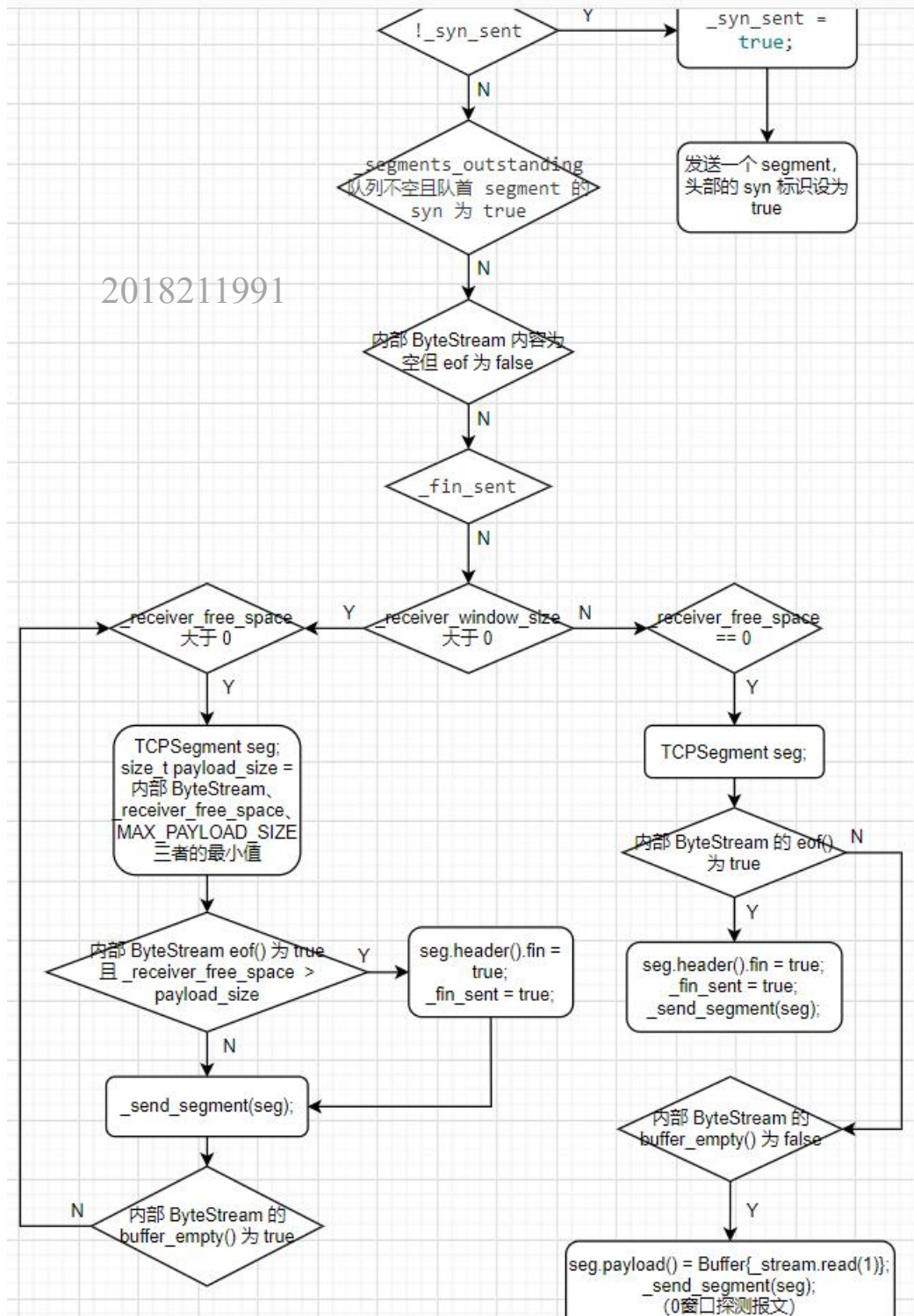


图 16 fill_window 函数流程图

```

void TCPSender::ack_received(const WrappingInt32 ackno, const uint16_t
window_size) {
    uint64_t abs_ackno = unwrap(ackno, _isn, _next_seqno);
    if (!ack_valid(abs_ackno)) {

```

```

        // cout << "invalid ackno!\n";
        return;
    }
    _receiver_window_size = window_size;
    _receiver_free_space = window_size;
    while (!_segments_outstanding.empty()) {
        TCPSegment seg = _segments_outstanding.front();
        if (unwrap(seg.header().seqno, _isn, _next_seqno) +
seg.length_in_sequence_space() <= abs_ackno) {
            _bytes_in_flight -= seg.length_in_sequence_space();
            _segments_outstanding.pop();
            // Do not do the following operations outside while loop.
            // Because if the ack is not corresponding to any segment in the
segment_outstanding,
            // we should not restart the timer.
            _time_elapsed = 0;
            _rto = _initial_retransmission_timeout;
            _consecutive_retransmissions = 0;
        } else {
            break;
        }
    }
    if (!_segments_outstanding.empty()) {
        _receiver_free_space = static_cast<uint16_t>(
            abs_ackno + static_cast<uint64_t>(window_size) -
            unwrap(_segments_outstanding.front().header().seqno, _isn,
_next_seqno) - _bytes_in_flight);
    }

    if (!_bytes_in_flight)
        _timer_running = false;
    // Note that test code will call it again.
    fill_window();
}

```

6. TCPConnection

该类的逻辑已在前文叙述的比较详尽，其主要工作是在内部实例化 TCPSender 和 TCPReceiver，并添加额外的逻辑，以正确地处理三次握手、四次挥手、以及在各个时间点发生意外的各种情况，最终建立一个稳健可用的 TCP 连接 peer 实例。

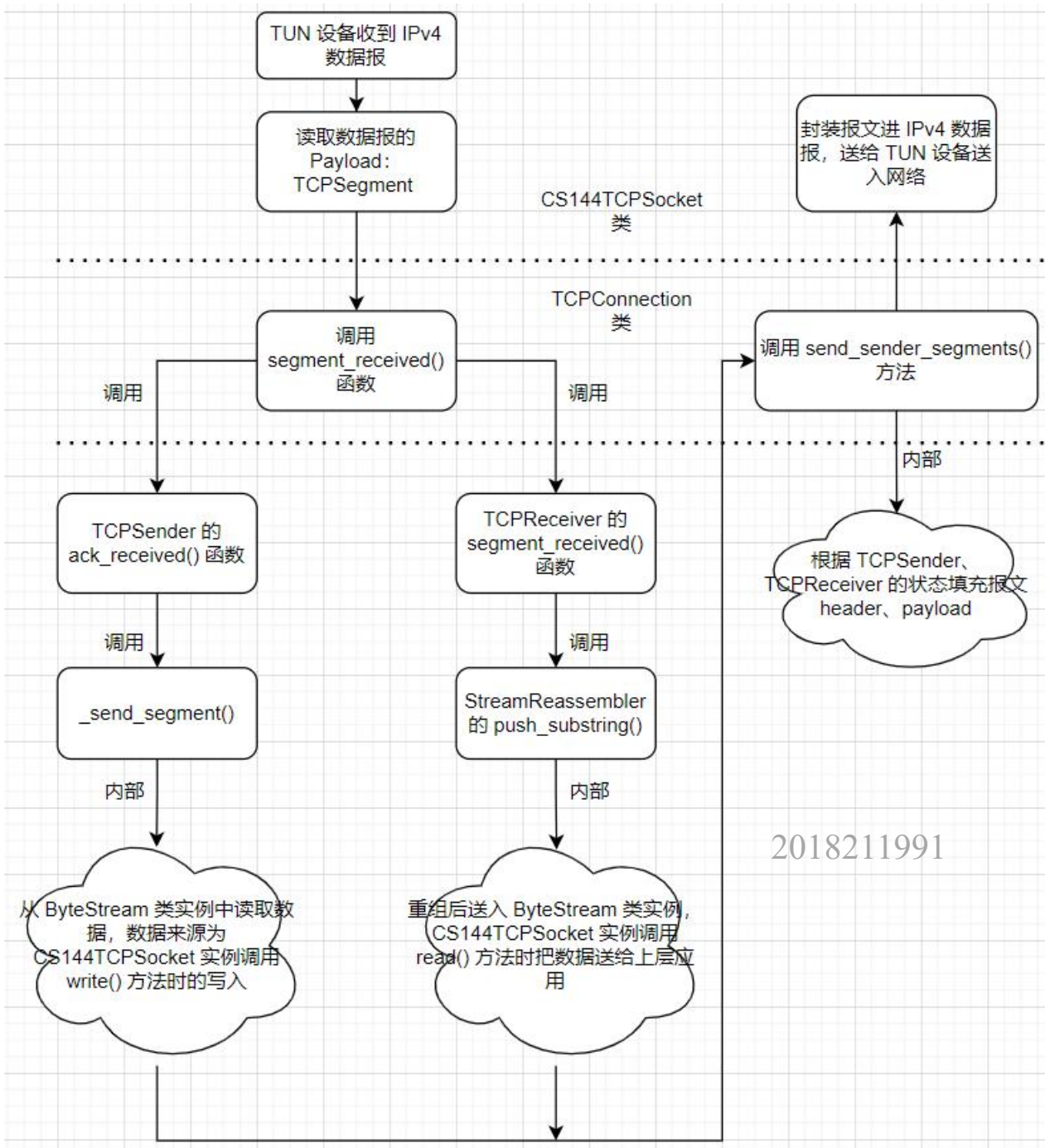


图 17 总流程图

```

tcp_connection.hh:
class TCPConnection {
private:
    size_t _time_since_last_segment_received{0};
    bool _active{true};

```



```
void send_sender_segments();
void clean_shutdown();
void unclean_shutdown();
```

tcp_connection.cc:

见源码

由于整个实验完备地考虑了 TCP 连接中的各种边缘情况，充斥着大量复杂琐碎的逻辑，许多函数流程图远比 fill_window 函数的流程图（图 16）更加复杂，并不能帮助我们有效地理解细节逻辑，本人书写的代码已经尽量做到自描述且在难以理解的地方加了必要的注释，因此不再给出更多的函数流程图。

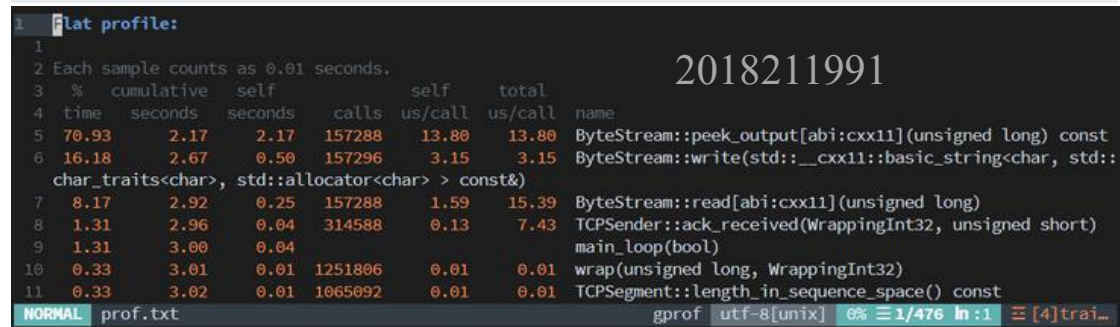
六、关键问题及其解决方法

实现过程中出现过大量的问题，基本上都是因为代码逻辑不够完善，没有正确的处理各种边缘情况。这里主要描述一下实现成功后的性能瓶颈和解决的过程。

初次完成实验后，本地性能仅为 0.2G/s，profiling 的第一步自然是定位瓶颈函数。

修改 /etc/cflags.cmake 中的编译参数，将 -g 改为 -Og -pg，使生成的程序具有分析程序可用的链接信息。

```
make -j8
./apps/tcp_benchmark
gprof ./apps/tcp_benchmark > prof.txt
```



% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
70.93	2.17	2.17	157288	13.80	13.80	ByteStream::peek_output[abi:cxx11](unsigned long) const
16.18	2.67	0.50	157296	3.15	3.15	ByteStream::write(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
8.17	2.92	0.25	157288	1.59	15.39	ByteStream::read[abi:cxx11](unsigned long)
1.31	2.96	0.04	314588	0.13	7.43	TCPSender::ack_received(WrappingInt32, unsigned short)
1.31	3.00	0.04				main_loop(bool)
0.33	3.01	0.01	1251806	0.01	0.01	wrap(unsigned long, WrappingInt32)
0.33	3.02	0.01	1065092	0.01	0.01	TCPSegment::length_in_sequence_space() const

图 18 profiling

可以看到，最大的问题出现在自底向上实现过程中的第一步 ByteStream 的实现，所以我们主要对 ByteStream 类进行优化。

在大致学习了 `std::move`, `assign`, 右值引用等 C++11 新特性，以及再次阅读 `util` 文件夹和 `tcp_helpers` 文件夹中的代码后，得出结论：调优方法是利用 `buffer.h` 中提供的 `BufferList`。

最初实现时我采用了 `std::list` 作为 ByteStream 内部的容器，这将会对参数进行多次拷贝销毁，而 `Buffer` 是基于引用计数实现的，其通过右值引用，基于内存所有权转移，避免了对数据的拷贝。

实际上测试代码中就有用到 `BufferList`，简而言之它是一个 `deque<Buffer>`，而 `Buffer` 则在整个实现与测试代码中被大量使用，例如 `payload()` 就是一个 `Buffer` 实例。

解决：

把 `ByteStream` 类中字节流的容器由最初的 `std::list<char> _stream{};` 改为 `BufferList _stream{};`。

`byte_stream.cc` 改动的函数：

```
size_t ByteStream::write(const string &data) {
    size_t write_count = data.size();
    if (write_count > _capacity - _buffer_size)
        write_count = _capacity - _buffer_size;
    _stream.append(BufferList(move(string()).assign(data.begin(), data.begin()
+ write_count))));
    _buffer_size += write_count;
    _bytes_written += write_count;
    return write_count;
}

///! \param[in] len bytes will be copied from the output side of the buffer
string ByteStream::peek_output(const size_t len) const {
    const size_t peek_length = len > _buffer_size ? _buffer_size : len;
    string str = _stream.concatenate();
    return string().assign(str.begin(), str.begin() + peek_length);
}

///! \param[in] len bytes will be removed from the output side of the buffer
```

```
void ByteStream::pop_output(const size_t len) {
    size_t pop_length = len > _buffer_size ? _buffer_size : len;
    _stream.remove_prefix(pop_length);
    _bytes_read += pop_length;
    _buffer_size -= pop_length;
}
```

调优后的性能：

```
/mnt/f/code/cpp/cs144/sponge/build lab4 !6 > ./apps/tcp_benchmark
CPU-limited throughput : 2.72 Gbit/s
CPU-limited throughput with reordering: 2.01 Gbit/s 2018211991
```

图 19 profiling 结果

七、设计结果

1. 所有测试样例的通过截图（包含各种几乎不可能发生的边缘情况）：

```
161/162 Test #163: t_isnD_128K_8K_L ..... Passed 0.52 sec
      Start 164: t_isnD_128K_8K_LL
162/162 Test #164: t_isnD_128K_8K_LL ..... Passed 0.83 sec

100% tests passed, 0 tests failed out of 162

Total Test time (real) = 58.02 sec
[100%] Built target check_lab4 2018211991

/mnt/f/code/cpp/cs144/sponge/build lab4 !1 > |
```

图 20 设计结果-1

2. 用我们自己编写的 TCPSocket，替代 Linux 给我们提供的 socket，与真实服务器进行通讯。

我们编写一个 webget 函数，向服务器发 HTTP get 请求。

```
void get_URL(const string &host, const string &path) {
    CS144TCPSocket sock1{};
    sock1.connect(Address(host, "http"));
    sock1.write("GET " + path + " HTTP/1.1\r\n" + "Host: " + host + "\r\n" +
"Connection: close\r\n\r\n");
```

```
while (!sock1.eof()) {  
    cout << sock1.read();  
}  
sock1.shutdown(SHUT_WR);  
sock1.wait_until_closed();  
}
```

结果:

```
~/code/cpp/sponge/build @38d0c25d > ./apps/webget cs144.keithw.org /hello  
HTTP/1.1 200 OK  
Date: Fri, 19 Mar 2021 04:35:43 GMT  
Server: Apache  
Last-Modified: Thu, 13 Dec 2018 15:45:29 GMT  
ETag: "e-57ce93446cb64"  
Accept-Ranges: bytes  
Content-Length: 14  
Connection: close  
Content-Type: text/plain  
2018211991  
Hello, CS144!
```

图 21 设计结果-2

```

~/code/cpp/sponge/build @38d0c25d > ./apps/webget www.baidu.com /
HTTP/1.1 200 OK
Accept-Ranges: bytes
Cache-Control: no-cache
Content-Length: 14615
Content-Type: text/html
Date: Fri, 19 Mar 2021 04:40:41 GMT
P3p: CP=" OTI DSP COR IVA OUR IND COM "
P3p: CP=" OTI DSP COR IVA OUR IND COM "
Pragma: no-cache
Server: BWS/1.1
Set-Cookie: BAIDUID=78BD737A6F3695C4FB251ED6D0CB7E4A:FG=1; expires=Thu, 31-Dec-37 23:59:59 GMT; max-age=2147483647; path=/; domain=.baidu.com
Set-Cookie: BIDUPSID=78BD737A6F3695C4FB251ED6D0CB7E4A; expires=Thu, 31-Dec-37 23:59:59 GMT; max-age=2147483647; path=/; domain=.baidu.com
Set-Cookie: PSTM=1616128841; expires=Thu, 31-Dec-37 23:55:55 GMT; max-age=2147483647; path=/; domain=.baidu.com
Set-Cookie: BAIDUID=78BD737A6F3695C4F817ED984598685C:FG=1; max-age=31536000; path=/; domain=.baidu.com
Traceid: 1616128841242318669810554362715590909443
Vary: Accept-Encoding
X-UA-Compatible: IE=Edge,chrome=1
Connection: close

<!DOCTYPE html><!--STATUS OK-->
<html>
<head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=Edge">
    <link rel="dns-prefetch" href="//s1.bdstatic.com"/>
    <link rel="dns-prefetch" href="//t1.baidu.com"/>
    <link rel="dns-prefetch" href="//t2.baidu.com"/>
    <link rel="dns-prefetch" href="//t3.baidu.com"/>
    <link rel="dns-prefetch" href="//t10.baidu.com"/>
    <link rel="dns-prefetch" href="//t11.baidu.com"/>
    <link rel="dns-prefetch" href="//t12.baidu.com"/>
    <link rel="dns-prefetch" href="//b1.bdstatic.com"/>
    <title>百度一下, 你就知道</title>
    <link href="http://s1.bdstatic.com/r/www/cache/static/home/css/index.css" rel="stylesheet">
    <!--[if lte IE 8]><style index="index" >#content{height:480px\9}#m{top:
    <!--[if IE 8]><style index="index" >#u1 a.mnav,#u1 a.mnav:visited{font-size:12px}
    <script>var hashMatch = document.location.href.match(/#+(.?wd=[^&].+)/);
    ?"+hashMatch[1]);}var ns_c = function(){};</script>

```

图 22 设计结果-3

八、软件使用说明

由于本课程设计属于底层设计而非展示应用，使用方法主要有以下两种：

1. 使用 build/app 目录下的可执行文件，注意需在 Linux 下使用，所有可执行文件均有 `-h` 参数来打印帮助信息，例如：

```
~/code/cpp/sponge/build lab4 > ./apps/tcp_ipv4 --help
Usage: ./apps/tcp_ipv4 [options] <host> <port>
```

Option		Default
2018211991		
-l	Server (listen) mode. In server mode, <host>:<port> is the address to bind.	(client mode)
-a <addr>	Set source address (client mode only)	169.254.144.9
-s <port>	Set source port (client mode only)	(random)
-w <winsz>	Use a window of <winsz> bytes	1452
-t <tmout>	Set rt_timeout to tmout	1000
-d <tundev>	Connect to tun <tundev>	tun144
-Lu <loss>	Set uplink loss to <rate> (float in 0..1)	(no loss)
-Ld <loss>	Set downlink loss to <rate> (float in 0..1)	(no loss)
-h	Show this message.	

```
ERROR: required arguments are missing.
```

图 23 使用说明-1

这些可执行文件也可以用于 Shell 脚本编程，例如：

```
get_cmdline_options "$@"

. "$(dirname "$0")/etc/tunconfig
REF_HOST=${TUN_IP_PREFIX}.144.1
TEST_HOST=${TUN_IP_PREFIX}.144.1
SERVER_PORT=$((RANDOM % 50000) + 1025))
if [ "$IUMODE" = "i" ]; then
    # IPv4 mode
    TEST_HOST=${TUN_IP_PREFIX}.144.9
    if [ -z "$USE_IPV4" ]; then
        REF_HOST=${TUN_IP_PREFIX}.145.9
        REF_PROG="./apps/tcp_ipv4 ${RTTO} ${WINSIZE} ${LOSS_UP} ${LOSS_DN} -d tun145 -a ${REF_HOST}"
        TEST_PROG="./apps/tcp_ipv4 ${RTTO} ${WINSIZE} -d tun144 -a ${TEST_HOST}"
    else
        REF_PROG="./apps/tcp_native"
        TEST_PROG="./apps/tcp_ipv4 ${RTTO} ${WINSIZE} ${LOSS_UP} ${LOSS_DN} -d tun144 -a ${TEST_HOST}"
    fi
else
    # UDP mode
    REF_PROG="./apps/tcp_udp ${RTTO} ${WINSIZE} ${LOSS_UP} ${LOSS_DN}"
    TEST_PROG="./apps/tcp_udp ${RTTO} ${WINSIZE}"
fi
```

图 24 使用说明-2

2. 对于网络编程程序员，可直接使用 CS144TCPSocket 类替代 Linux 内核的 TCPSocket 类进行编程，如第七部分设计结果中的 webget 函数：


```

CS144TCPSocket sock1{};
sock1.connect(Address(host, "http"));
sock1.write("GET " + path + " HTTP/1.1\r\n" + "Host: " +
while (!sock1.eof()) {
    cout << sock1.read();
}
sock1.shutdown(SHUT_WR);
sock1.wait_until_closed();
2018211991

```

图 25 使用说明-3

正如前文所述，我们在 TCPConnection 类基础上封装的 CS144TCPSocket 与 UNIX 标准的 socket api 别无二致。

唯一的区别在于，我们的 socket 实例需要在最后调用 wait_until_closed() 函数。这是因为，通常情况下，即使在用户进程退出后，Linux 内核也会负责等待 TCP 连接达到“干净地释放连接”（并放弃它们的端口保留）。但是因为我们的 TCP 实现在用户态中，所以除了我们的程序进程之外，没有其他东西可以跟踪连接状态。添加此调用会使 socket 等待，直到我们的 TCPConnection 报告 active() = false，这是安全的结束程序的方式。

九、参考资料

- [1] CS 144: Introduction to Computer Networking, Fall 2020.
<https://cs144.github.io>
- [2] 个人博客笔记. <https://segmentfault.com/u/wine99/articles>
- [3] TCP State Transitions.
<http://ttcplinux.sourceforge.net/documents/one/tcpstate/tcpstate.html>
- [4] TUN/TAP 设备浅析. <https://www.jianshu.com/p/09f9375b7fa7>

十、验收时间及验收情况

2021 年 3 月 18 日下午。

十一、设计体会

1. 收获体会

本次课程设计的部分工作于寒假完成，总花费时间大约为集中的 9 天加上零

零碎碎的 6 天，最后的 TCPConnection 刚开始时完全无从下手，一度想要放弃。但最终通过了所有测试，完整的实现了 TCP 连接的整个逻辑，非常有成就感。此次实验也我对 TCP 协议有了更深的认识，尤其是握手挥手等过程的细节。

在实现 TCPConnection 类时，我浏览了国内外网友的三四个实现方式，发现无一例外地改动了前面实现的类的部分函数签名和函数逻辑（也就是说为了成功地封装 Sender 和 Receiver，改动了这两个类的设计），这让整个实现变得不太干净。我的最终实现没有入侵 Lab3 和 Lab2 的代码，细节逻辑全部在 TCPConnection 类中完成。这也是出乎我意料之外的成就。

同时，此次实验也让我认识到了自己对于语言掌握的不足。即使做完了整个项目，仍然对 C++ 有所畏惧。整个项目没有任何一处内存不安全的代码，正如原课程实验的要求：使用现代 C++，例如禁止使用 new、delete、原始指针（实际上智能指针也没有用到）、C 风格字符串，尽量使用引用传参等等。同时我也了解到了一些 C++ 的高级技巧，例如右值引用，move 函数等等，在性能优化阶段，使用这些新特性使性能直接从 0.2G/s 提高到了 2.7G/s。

2. 不足之处

可以继续向底层扩展：本次课程设计做到了利用 TUN 设备发送自己的 IPv4 数据包，payload 为自己的 TCP 报文段，可以更进一步，利用 TAP 设备，自己实现 ARP 协议、最长前缀匹配以及链路层数据帧。（TUN 是三层模拟设备，不具备自己的 MAC 地址，TAP 是二层模拟设备，模拟到了链路层，在协议栈的眼中，tapX 和真实网卡没有任何区别。）

3. 补充信息

实现代码已托管 Github: <https://github.com/wine99/cs144-20fa>。

与项目相关的博客已在网上发布，报告中的部分雷同并非抄袭，链接见第九部分参考资料。