* [首页](https://km.woa.com/?kmref=km_header)
* [发现](https://km.woa.com/discovery?kmref=km_header)
* [悦读](https://km.woa.com/read?kmref=km_header)
* [乐问](https://km.woa.com/q?kmref=km_header)
* [直播](https://tencent.lexiangla.com/lives?company_from=tencent&kmref=km_header)
* [应用](javascript:void(0))
* [我的K吧](javascript:void(0))
* [[](https://km.woa.com/user/rickyrqzhao)](https://km.woa.com/user/rickyrqzhao)

窗体顶端



窗体底端

[](https://km.woa.com/group/tsign)[腾讯电子签技术部落](https://km.woa.com/group/49894)

* [首页](https://km.woa.com/group/49894)
* [团队微博](https://km.woa.com/group/49894/twitters)
* [团队讨论](https://km.woa.com/group/49894/topics)
* [团队文章](https://km.woa.com/group/49894/articles)
* [团队文档](https://km.woa.com/group/49894/docs)
* [团队活动](https://km.woa.com/group/49894/events)
* [团队投票](https://km.woa.com/group/49894/surveys)
* [团队日历](https://km.woa.com/group/49894/calendars)
* [团队会议](https://km.woa.com/group/49894/meetings)

目录

TCP 概述

TCP 简单介绍

面向连接

可靠性

基于字节流

全双工

Sponge 协议介绍

Socket API

用户态协议

Sponge 协议概览

Sponge 协议实现

ByteStream

StreamReassembler

TCPReceiver

TCPSender

TCPConnection

Sponge 使用

总结

参考资料

[[](https://km.woa.com/base/headlines)](https://km.woa.com/base/headlines" \t "_blank)

谈谈用户态 TCP 协议实现

* [pedrogao](https://km.woa.com/user/pedrogao)

* 2021年09月02日 15:22

* 浏览(1178)

* [收藏(104)](javascript:void(0);)

* [评论(9)](javascript:void(0);)

* [分享](javascript:void(0);)

| 导语 TCP 协议是目前名气最大、使用最广泛的传输层网络协议。 TCP 是一个可靠的（reliable）、面向连接的（connection-oriented）、基于字节流（byte-stream）、全双工的（full-duplex）协议。 正是因为这些优点，TCP 协议成为了网络协议重点中的重点，是学习、面试、考试上的常客，这也导致了 TCP 的资料很多，但是普遍集中在“形”上面，很多人将三次握手、四次挥手、滑动窗口等知识点背得滚瓜烂熟，但却没有理解 TCP “可靠” 协议的精髓。

**TCP 概述**

TCP 协议是目前名气最大、使用最广泛的传输层网络协议。

TCP 是一个可靠的（reliable）、面向连接的（connection-oriented）、基于字节流（byte-stream）、全双工的（full-duplex）协议。

正是因为这些优点，TCP 协议成为了网络协议重点中的重点，是学习、面试、考试上的常客，这也导致了 TCP 的资料很多，但是普遍集中在“形”上面，很多人将三次握手、四次挥手、滑动窗口等知识点背得滚瓜烂熟，但却没有理解 TCP “可靠” 协议的精髓。

因此，本着实践加深理解的初衷，笔者跟随 CS144 这门课，会从头到尾实现了一个用户态简易版 TCP 协议。

说明：CS144 最终会实现一个全栈的网络协议栈，包含数据链路层、网络层和传输层，但是本文的重点聚焦在 TCP 协议上，因此对于其它层协议不做详细介绍，感兴趣的可以自行查询。

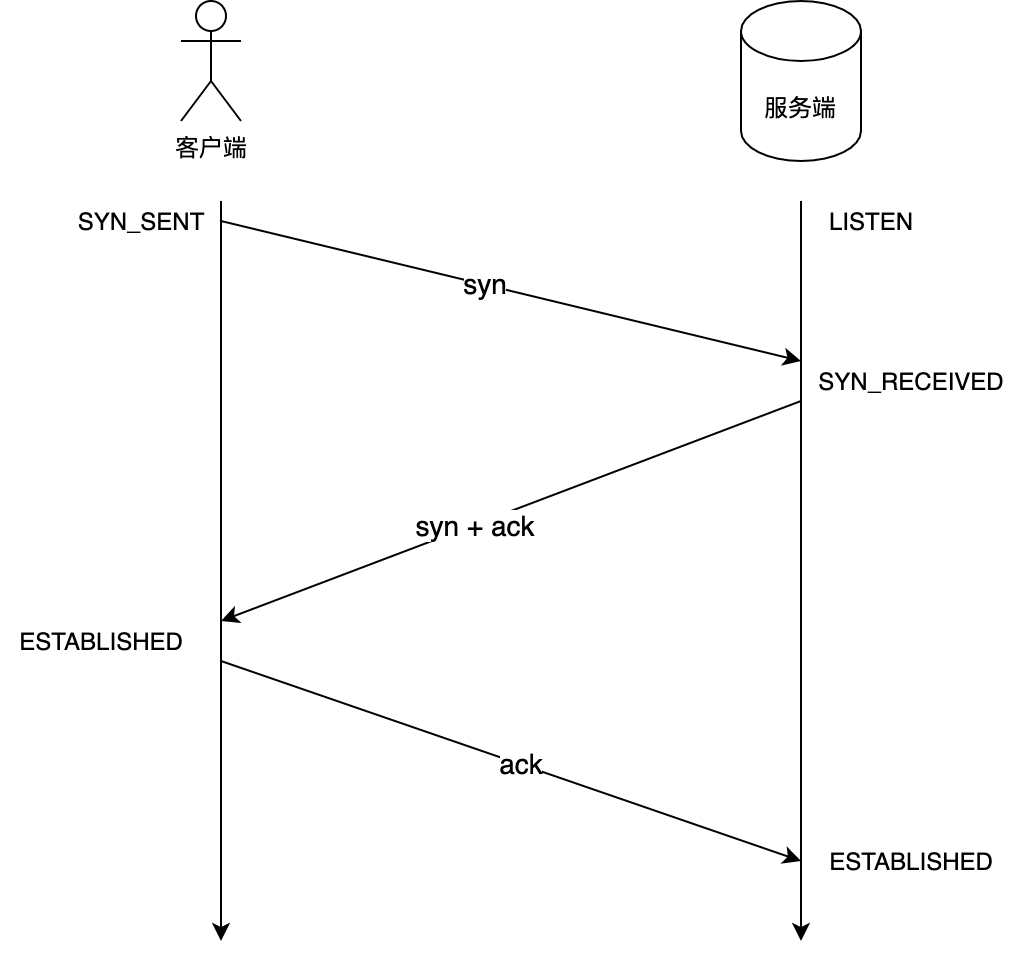
**TCP 简单介绍**

下面我们分别简单介绍一下 TCP 的特点。

**面向连接**

面向连接是 TCP 显著特点，在正式数据传输之前 TCP 需要三次握手来协商建立连接，结束传输后又需要四次挥手来结束连接。

以三次握手为例，TCP 需要通过三次握手来确认对端状态，交换起始序号、窗口大小等信息，如下：



三次握手流程如下：

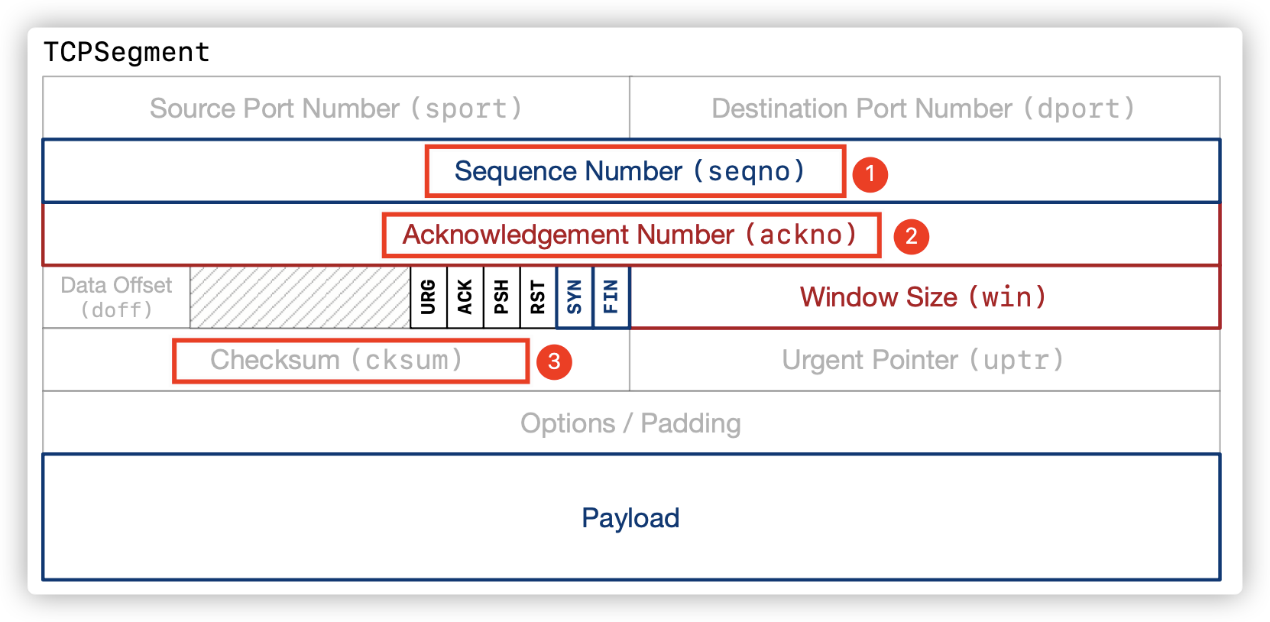
1. 客户端向服务器发送 SYN 包；
2. 服务器收到 SYN 包后，向客户端发送 SYN+ACK 包；
3. 客户端收到 SYN+ACK 包后，回复 ACK 包至客户端。

在三次握手的过程中，重点在于 SYN 和 ACK 包的交互，当然也涉及到初始化序号、窗口大小、状态转换等工作，这些细节后面会在实现中详细介绍。

**可靠性**

可靠性是 TCP 最大的功能点(个人观点，欢迎斧正)，TCP 为了保证数据传输的可靠性，做了很多事情，虽然这增加了实现的复杂性，但却是值得的：

* 校验和，TCP 每个报文都有校验和字段，防止数据丢失或出错；
* 序列化和确认号，保证每个序号的字节都交付，解决丢失、重复等问题；
* 超时重传，对于超时未能确认的报文，TCP 会重传这些包，确保数据达到对端；
* 等等

如下： 

虽然 TCP 在可靠性上做了很多努力，但仍然不能保证完美的可靠性，只能做到尽最大努力交付。对于可靠性的细节，我们将在后面的实现中详细介绍。

**基于字节流**

TCP 数据传输是基于流的，意味着 TCP 传输数据是没有边界的，没有大小限制的，可以传输无限量的字节。

但是 TCP 报文大小是有限制的，这主要取决于滑动窗口大小、路径最大传输单元 MTU 等因素。

TCP 数据写、读、传入都是基于字节流，因此常常会有字节流乱序发生，所以 TCP 需要一个重组器组件专门用于流序号的重组工作，当然这涉及到 TCP 具体实现，我们将实现部分详细介绍。

**全双工**

全双工意味着 TCP 协议通信的双方既可以发送数据，又可以接受数据，双方拥有独立的序号、窗口等信息。

简单来说，一个 TCP 连接既可以是 sender 也可以是 receiver，同时连接拥有两个字节流，一个输出流，被 sender 控制，另一个是输入流，由 receiver 管理。关于这二者的细节，我们将在实现部分详细介绍。

**Sponge 协议介绍**

**Socket API**

绝大多数操作系统会在内核中提供 TCP 协议的实现，并对外暴露 socket API，借助 nc 工具，我们可以快速的使用 socket API，并领略它的风采，如下：

$ nc -v -l -p 9090

Ncat: Version 6.40 ( http://nmap.org/ncat )

Ncat: Listening on :::9090

Ncat: Listening on 0.0.0.0:9090

首先，在一个窗口中通过 -l 参数来监听本地的 9090 端口，然后在另一个窗口中连接该端口：

$ telnet localhost 9090

回到 nc 端口，会发现多出了如下日志：

Ncat: Connection from 127.0.0.1.

Ncat: Connection from 127.0.0.1:37731.

可以看到，通过 nc 和 telnet 这两个工具，我们很轻易的就建立起了 TCP 连接，而这两个工具本身都是在调用 socket API。 TCP 由操作系统实现在了内核态，并提供 socket API，虽然使用方便，但是却屏蔽了大量信息，给错误调试和定位带来了很大困难。

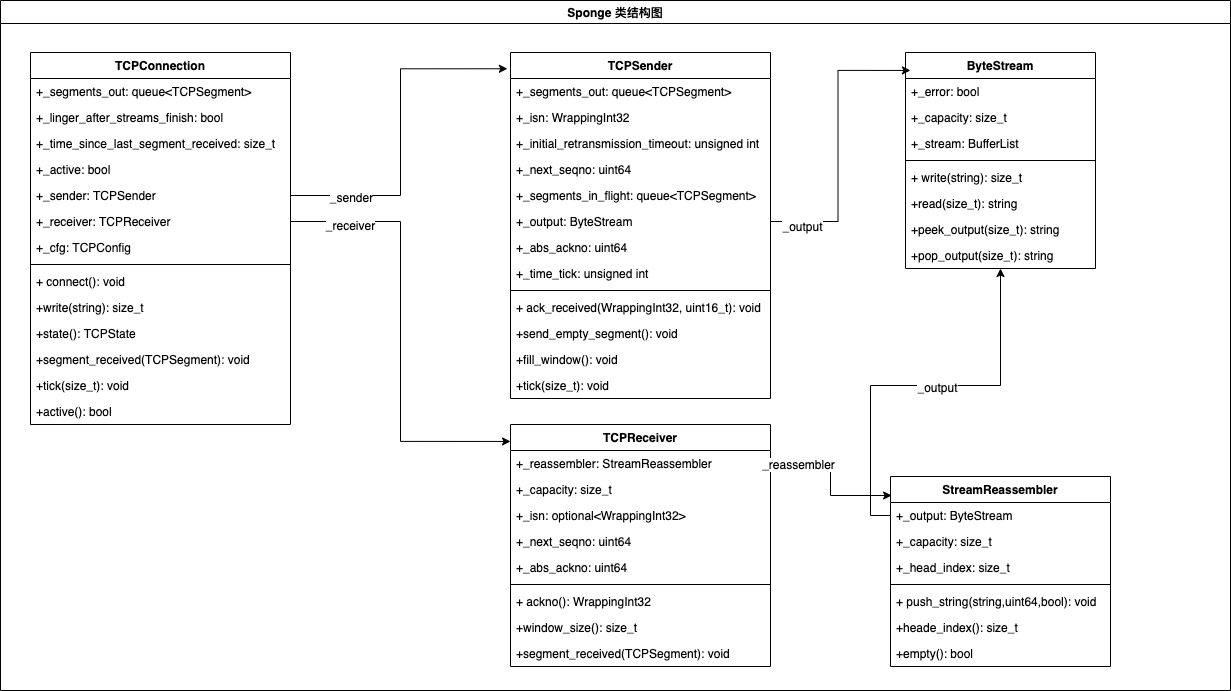
**用户态协议**

在用户态上实现协议是一个很有趣且很有挑战的事情，目前最著名的当属 QUIC 协议，QUIC 协议是建立在 UDP 上的一个用户态可靠协议。而 CS144 实现的 TCP 协议，也就是本文后面所实现的 TCP 协议和 QUIC 协议定位非常类似，当然 QUIC 协议在功能特性上完胜，但是这不影响我们以此来学习一个简单可靠的类 TCP 协议。

为了与其它协议进行区分，本文实现的协议我们统称为 Sponge 协议。下面，对 Sponge 协议做一些简单的介绍：

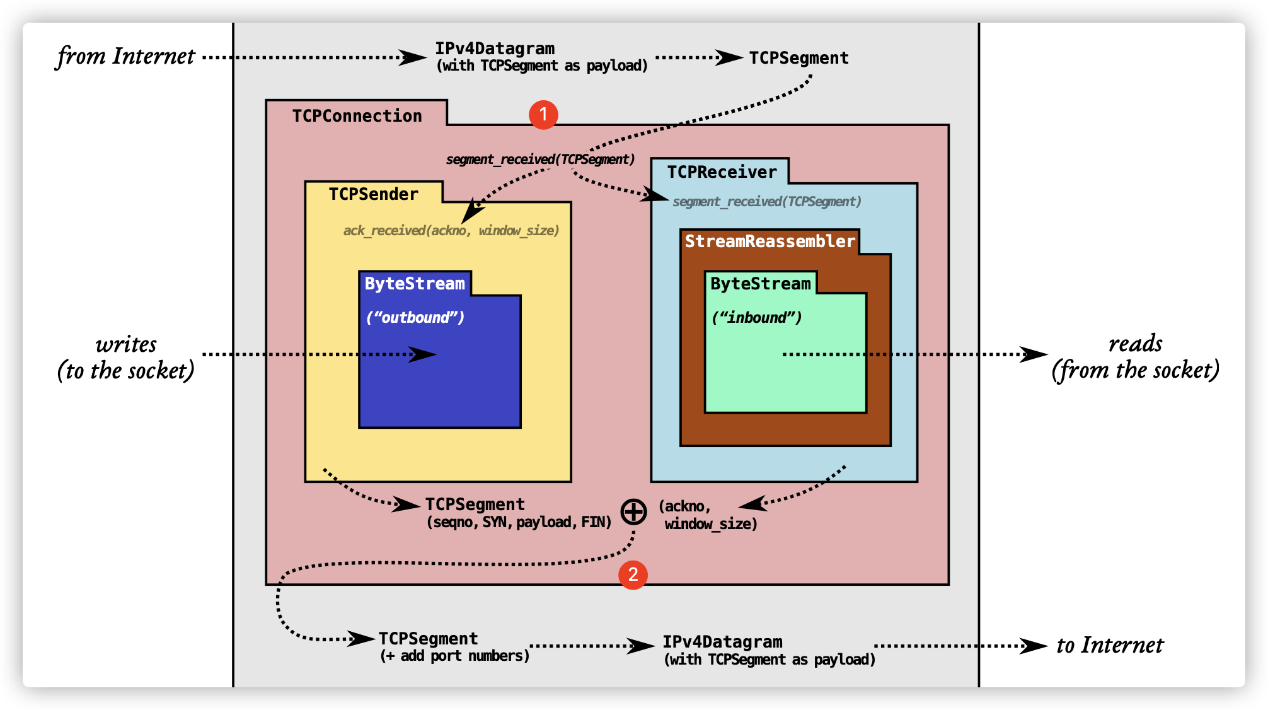
1. Sponge 协议建立在 UDP 之上(也可以建立在 IP 协议之上，为了避免引入 TUN/TAP 带来复杂，暂时不做延伸)；
2. Sponge 协议是一种简易版 TCP 协议，和 TCP 协议一样有滑动窗口、重传、校验和等功能，但是一些复杂的特性暂时不支持，如：紧急指针、拥塞控制、Options 中的一些选项均不支持；
3. Sponge 协议并不特别复杂，在 CS144 的课程带领下，完全可以自主实现，并没有什么高深莫测的技术，每个人都有能力去理解和实现。

**Sponge 协议概览**

下面，我们就以先整体后局部的方式来详细介绍 Sponge 协议。Sponge 协议的主体类为 TCPConnection，该类主要维护 TCP 连接、TCP 状态机等信息数据，并将接收到的报文交给 TCPReceiver 处理，从 TCPSender 获取报文并发送。类图如下： 

* TCPConnection 负责维护连接，报文发送、接收分别由 TCPSender 和 TCPReceiver 来负责；
* TCPSender 负责发送报文，接收确认号(ackno)确认报文，记录发送但未确认的报文，对超时未确认的报文进行重发；
* TCPReceiver 负责接收报文，对报文数据进行重组(报文可能乱序、损坏等，由 StreamReassembler 负责重组)；
* StreamReassembler 负责对报文数据进行重组，每个报文中的每个字节都有唯一的序号，将字节按照序号进行重组得到正确的字节流，并将字节流写入到 ByteStream 中；
* ByteStream 是 Sponge 协议中的字节流类，一个 TCPConnection 拥有两个字节流，一个输出流，一个输入流。输出流为 TCPSender 中的 \_output 字段，该流负责接收程序写入的数据，并将其包装成报文并发送，输入流为 StreamReassembler 中的 \_output 字段，该流由 StreamReassembler 重组报文数据而来，并将流数据交付给应用程序。

Sponge 协议巧妙地将连接分为了 sender 和 receiver 两个部分，并通过重组器、字节流等类将 TCP 连接完美抽象，使代码更易维护和阅读，也使功能迭代和完善更加方便。

Sponge 的数据流图如下所示： 

从这个图中可以总结出 Sponge(基于 UDP 而非 IPv4) 数据流过程：

1. 内核态下 UDP 数据包中的 payload 被解析为 TCPSegment(TCP 报文)后，交给用户态下的 TCPConnection，即调用 segment\_received 方法；
2. TCPConnection 收到报文后，将报文交给 TCPReceiver，即调用 TCPReceiver.segment\_received 方法，并将报文中的 ackno(确认号)与 window\_size(窗口大小)交给 TCPSender，即调用 ack\_received 方法；
3. TCPReceiver 处理 TCP 报文，并将报文中的 payload 推入 StreamReassembler 中，并重组后交给应用程序，随后尝试发送报文；
4. TCPConnection 调用 TCPSender.fill\_window 方法尝试得到待发送报文(可能得不到，视具体情况而定)，若有报文，则设置报文 payload 以及其它字段，如 SYN、ackno(从 receiver 获取)、window\_size 等，设置完毕后包装为 TCP 报文，将报文交给 UDP；
5. UDP 将其打包为数据报，并发送给远端。

**Sponge 协议实现**

在粗略介绍 Sponge 协议后，我们一起来看看 Sponge 协议的具体实现逻辑和一些细节。

在介绍 Sponge 协议时，采用的是先整体后局部的方式，而在说明具体实现时，需要先从局部出发，逐渐上升到整体，因此我们会先从 ByteStream 数据流开始，逐步添砖加瓦最后实现一个完成的 TCPConnection。

这里先给出 Sponge 协议代码核心文件：

libsponge/

├── byte\_stream.cc // ByteStream(数据流) 实现文件

├── byte\_stream.hh // ByteStream 头文件

├── stream\_reassembler.cc // StreamReassembler(数据流重组器) 实现文件

├── stream\_reassembler.hh // StreamReassembler 头文件

├── tcp\_connection.cc // TCPConnection(TCP连接) 实现文件

├── tcp\_connection.hh // TCPConnection 头文件

├── tcp\_receiver.cc // TCPReceiver(TCP接收者) 实现文件

├── tcp\_receiver.hh // TCPReceiver 头文件

├── tcp\_sender.cc // TCPSender(TCP发送者) 实现文件

├── tcp\_sender.hh // TCPSender 头文件

├── wrapping\_integers.cc // WrappingIntegers(包装32位seqno、ackno)实现文件

└── wrapping\_integers.hh // WrappingIntegers 头文件

**ByteStream**

ByteStream 是一个基于内存、可靠的数据流实现类。Sponge 协议类似于 TCP 协议，也是一个基于字节流的网络协议，数据流是协议最核心的数据载体，在上面也谈到了，一个 TCPConnection 有两个数据流，分别用于数据输入和输出。

ByteStream 是一个可读且可写且有容量限制的数据流。ByteStream 在初始化时，会被设置一个 capacity(容量) 参数，表示该数据流不能存储超过容量的字节数。ByteStream 实现是比较简单的，既可以直接使用 string(字符串)来实现，也可以使用 sponge 项目提供的 BufferList 来实现。

推荐使用 BufferList，内部通过共享指针来实现字符串容器，减少字符串拷贝带来的性能、内存损耗。

这里摘出 ByteStream 中几个重要的实现方法加以说明：

ByteStream::ByteStream(const size\_t capacity) : \_capacity(capacity) {}

size\_t ByteStream::write(const string &data) {

size\_t sz = data.size();

// 如果数据超过了窗口大小，那么等于窗口大小

if (sz > \_capacity - \_buffer\_size) {

sz = \_capacity - \_buffer\_size;

}

\_stream.append(BufferList(move(string().assign(data.begin(), data.begin() + sz))));

\_buffer\_size += sz;

\_bytes\_written += sz;

return sz;

}

string ByteStream::peek\_output(const size\_t len) const {

// peek 的长度不能超过 buffer\_size

const size\_t peek\_length = len > \_buffer\_size ? \_buffer\_size : len;

string str = \_stream.concatenate();

return string().assign(str.begin(), str.begin() + peek\_length);

}

void ByteStream::pop\_output(const size\_t len) {

// 不能超过 buffer\_size

size\_t pop\_length = len > \_buffer\_size ? \_buffer\_size : len;

\_stream.remove\_prefix(pop\_length);

\_bytes\_read += pop\_length;

\_buffer\_size -= pop\_length; // 更新 buffer\_size

}

std::string ByteStream::read(const size\_t len) {

const string result = peek\_output(len);

pop\_output(len);

return result;

}

这段代码展示了 ByteStream 中最重要的 5 个方法：

* ByteStream 构造函数，用于初始化一个字节流；
* write 函数，向字节流中写入数据，注意写入数据的大小和当前缓冲区的大小加起来不能超过容量大小，然后将数据加入到 \_stream 容器中，并且更新 buffer\_size 和 bytes\_written；
* peek\_output 函数，查看字节流的前 len 个字节，peek\_out 方法不会消费字节流，只会查看前 len 个字节，并且查询字节数量不能超过当前缓冲区字节的数量；
* pop\_out 函数，移除字节流中的前 len 个字节，然后更新 bytes\_read 和 buffer\_size；
* read 函数，读取字节流中的前 len 个字节，注意 read 会消费流数据，读取后会移除前 len 个字节。

ByteStream 是 spong 项目中最简单的一部分，但也是最容易忽略的一部分，笔者最开始在实现的时候小觑了这里，以为通过了全部单测就没有问题了，但是在后面的测试中，发现数据有错误，debug 了好久才察觉是 ByteStream 有问题。

**StreamReassembler**

一个 TCP 连接只有一个 StreamReassembler，用于 receiver 重组乱序的报文数据。由于 TCP 报文可能存在乱序、重复、缺少等各种各样的问题，receiver 收到的 TCP 报文数据不能直接写入到 ByteStream 中交付给应用程序。

因此为了解决 TCP 报文数据乱序等问题，sponge 将重组这部分工作抽象为数据流重组器。由于网络延迟、丢包等场景，比如：

1. 发送 A、B 包，结果 B 包比 A 包更早到达，因此 B 包到达时，不可直接写入字节流；
2. 发送 A、B 包，接收 A 包丢了，只有 B 包达到，那么 B 包到达后也不可直接写入字节流，必须等待 A 包重发达到后。

为了解决这些问题，我们必须有序的组装 TCP 报文，如下：

class StreamReassembler {

private:

ByteStream \_output; //!< The reassembled in-order byte stream

size\_t \_capacity; //!< The maximum number of bytes

size\_t \_head\_index; //!< 当前序号

// 重组节点

struct node {

size\_t begin = 0;

size\_t size = 0;

std::string data{};

// 重载 < 符号，用于序列重组

bool operator<(const node &a) const { return begin < a.begin; }

};

// 节点合并

long merge(node &n1, const node &n2);

std::set<node> \_set{}; // 数据容器，二叉树

size\_t \_unassembled\_bytes; // 未组装字节个数

}

我们在 StreamReassembler 的定义中添加了 \_head\_index 和 \_set 字段：

* \_head\_index：当前已组装的字节序号，默认为 0，即没有字节被重组；
* \_set：TCP 报文容器，每个 TCP 报文都被当作一个 node，包含了数据、序号等信息，node 存储在 set 中，set 内部是一个红黑树实现，支持节点有序化。

当一个报文数据被推入到重组器后，重组器判断报文头字节序号与 \_head\_index 之间的关系，对数据进行裁剪，得到当前数据节点，由于 set 中的 node 是按照 index 有序排列，寻找节点尝试合并数据重复的节点，合并完毕后，判断节点 index 与 \_head\_index 之间的关系，将 index 刚好等于 \_head\_index 的节点数据写入字节流中。如下：

void StreamReassembler::push\_substring(const string &data, const size\_t index, const bool eof) {

// 接收到 data 后，将新的连续的字节写入到流中

// 如果超过了 capacity 那么丢弃掉数据

// 如果 eof 为 true，那么接收的为最后一个字符串

size\_t sz = data.size();

if (eof) {

this->\_eof = eof;

}

// 如果 sz == 0 或者 data 已经被重组了，那么直接返回

if (sz == 0 || sz + index < \_head\_index) {

handle\_eof();

return;

}

node cur;

if (index < \_head\_index) {

// 当前 index 小于 \_head\_index

// 那么截取 index 后面有用的部分

size\_t offset = \_head\_index - index;

cur.begin = \_head\_index;

cur.size = sz - offset;

cur.data = data.substr(offset);

} else {

// 当 index >= \_head\_index

cur.begin = index;

cur.size = sz;

cur.data = data;

}

\_unassembled\_bytes += cur.size; // 未重组

// 如果节点之间有数据重合，那么依次合并节点

long total = 0;

// 向后合并

while (true) {

// 找到第一个 >= cur.begin 的节点

auto next = \_set.lower\_bound(cur);

if (next == \_set.end()) {

break; // 没有找到

}

// next 是第一个 >= cur 的节点

// 合并 cur 和 next 到 cur

long merged\_count = merge(cur, \*next);

if (merged\_count < 0) {

break; // 小于 0，证明无法合并，直接 break

}

total += merged\_count;

// 合并 next 后然后删除 next，注意所有数据都被合并到了 cur 节点中

\_set.erase(next);

}

// 向前合并

auto next = \_set.lower\_bound(cur);

while (next != \_set.begin()) {

next--; // 得到前一个节点

long merged\_count = merge(cur, \*next);

if (merged\_count < 0) {

break;

}

total += merged\_count;

\_set.erase(next);

next = \_set.lower\_bound(cur); // 找到下一个

}

// 向前、向后一起合并了 total

\_unassembled\_bytes -= total;

// 插入 cur

\_set.insert(cur);

// 合并完成以后，再来推进 \_head\_index

while (!\_set.empty() && \_set.begin()->begin == \_head\_index) {

auto begin = \_set.begin();

size\_t written = \_output.write(begin->data);

\_head\_index += written;

\_unassembled\_bytes -= written;

\_set.erase(begin); // 删除已经合并的头节点

}

handle\_eof();

}

上面代码已经对节点合并、\_head\_index 推进给出了详细注释。push\_substring 是 StreamReassembler 中最复杂、最重要的函数，它接受三个参数：

* data：报文数据(不含 TCP header)；
* index：报文数据第一个字节的序号，注意是字节流的序号，跟 seqno 有区别，后面再细说；
* eof：是否收到了 fin 包数据，即是否要关闭输入数据流。

由于 set 容器中节点的有序性(node 重载了< 保证了 begin 小的节点在前面)，可通过 lower\_bound 函数直接找到第一个大于等于当前节点 begin 的节点，然后尝试去合并二者。

StreamReassembler 另外一个需要注意的点是何时关闭输入流，如下：

void StreamReassembler::handle\_eof() {

if (\_eof && empty()) {

// 如果标记了结束，且为空

\_output.end\_input();

}

}

当收到 eof 为 true 时，不能立马关闭 \_output，因为 fin 包可能先于其它数据包达到，因此我们必须缓存 eof，等待重组器中数据为空时再关闭输入流。

**TCPReceiver**

TCPReceiver 是 TCP 连接的接收方，负责接收报文，并将报文数据(payload)交给 StreamReassembler 重组，得到正确的字节流序列。定义如下：

class TCPReceiver {

//! Our data structure for re-assembling bytes.

StreamReassembler \_reassembler;

//! The maximum number of bytes we'll store.

size\_t \_capacity;

// 省略一些字段

std::optional<WrappingInt32> \_isn{std::nullopt};

// absolute ack number

uint64\_t \_abs\_ackno;

}

* \_reassembler：数据流重组器；
* \_capacity：容量，用于 \_reassembler 初始化；
* \_isn：收到的第一个 seqno；
* \_abs\_ackno：绝对确认序号。

这里需要额外说明一点：关于 WrappingInt32(seqno,ackno) 与 uint64\_t(abs\_seqno,abs\_ackno)。

在 TCP header 中 seqno 与 ackno 都是 32 位整数，但是 isn 是发送者随机生成的数字，那么可能会产生一个问题：如果随机生成的 isn 十分接近 2^32-1 怎么办？

这代表，TCP 数据流很快就会溢出 seqno。为了解决这个问题，sponge 实现了两个序号，一个是正常的 TCP header seqno，32 位，使用 WrappingInt32 表示，初始化为随机数，另一个是绝对 seqno 序号，使用 uint64\_t 表示，初始化为 0。用绝对序号来表示数据流上字节的数量，最大数量为 2^64-1，这几乎是不可能达成的目标。

另外在 StreamIndex 中的 push\_string 函数有一个 index 参数，index 表示字节在数据流中的序号，那么这个序号与绝对 seqo 序号，以及 seqno 有什么联系呢？

假设 isn = 2^32−2 且向数据流中写入 "cat" 三个字节，那么 seqno、abs\_seqno、stream\_index 之间的对应关系如下表：

| **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 |  |

* seqno 溢出后，重新从 0 开始；
* absolute\_seqno 从 0 开始，并与 seqno 一一对应；
* syn 和 fin 均占一个序列；
* stream\_index 从 0 开始，但是不能包括 syn 与 fin。

实现 TCPReceiver 的核心函数有三个，如下：

void TCPReceiver::segment\_received(const TCPSegment &seg) {

// Corner Case:

// 如果包头部没有 syn，且 receiver 也没有 \_syn 过，那么直接返回

// 第一个接收的包一定包含了 syn

WrappingInt32 seqno = seg.header().seqno;

if (!seg.header().syn && !\_syn) {

return;

}

bool first\_syn = false;

// 第一次收到 syn 包：

// 设置 \_isn 和 \_syn

if (seg.header().syn && !\_syn) {

\_isn = std::make\_optional(seqno);

\_syn = true;

first\_syn = true;

}

// 收到 fin 包，且本地没有 \_fin

if (seg.header().fin && !\_fin) {

\_fin = true;

}

Buffer payload = seg.payload();

// 得到 seqno 对应的 abs\_seqno

// In your TCP implementation, you’ll use the index of the last reassembled byte as the checkpoint.

// checkpoint 为最后一个重组的字节序号，即 head\_index

uint64\_t abs\_seqno = unwrap(seqno, \_isn.value(), \_reassembler.head\_index());

// 如果该 seqno 已经被接收过了，那么本次将忽略

// 如果超过了窗口阈值，那么也无法接收

if (abs\_seqno + seg.length\_in\_sequence\_space() <= \_abs\_ackno || abs\_seqno >= \_abs\_ackno + window\_size()) {

return;

}

size\_t stream\_index = abs\_seqno; // stream\_index 默认为 abs\_seqno

string data = payload.copy();

// 如果 syn 也伴随着数据，那么 abs\_seqno 应该等于 1

if (\_syn && !first\_syn && stream\_index > 0) { // 如果 syn 那么 -= 1, syn 不算

stream\_index -= 1;

}

// fin 会占有 seqno 序列，但是前提是内部已经关闭，数据流上达到了 fin

if (\_fin && \_reassembler.stream\_out().input\_ended() && stream\_index > 0) { // 如果 fin，那么 -= 1, fin 不算

stream\_index -= 1;

}

// 推入数据

\_reassembler.push\_substring(data, stream\_index, seg.header().fin);

// 更新 \_abs\_ackno

// 这个地方的 +1 直接抵消了 syn 包

\_abs\_ackno = \_reassembler.head\_index();

if (\_syn) {

\_abs\_ackno += 1;

}

// fin 且数据流关闭，才能确认 fin，这才是 fin 的真正关闭条件

// 即 FIN\_RECV = stream\_out().input\_ended()

if (\_fin && \_reassembler.stream\_out().input\_ended()) {

\_abs\_ackno += 1;

}

return;

}

optional<WrappingInt32> TCPReceiver::ackno() const {

// If the ISN hasn’t been set yet, return an empty optional

if (!\_isn.has\_value()) {

return std::nullopt;

}

// 注意：fin 包也占有一个序号

// This is the windows’s left edge: the first byte the receiver is interested in receiving.

// \_abs\_ackno 已经在 segment\_received 函数中做好了计算，直接 wrap 即可

return wrap(\_abs\_ackno, \_isn.value());

}

size\_t TCPReceiver::window\_size() const {

// the distance between the “first unassembled” index (the index corresponding to the

// ackno) and the “first unacceptable” index.

// 已经确认的序列-未收到的序列

// 这里的 window\_size 应该是 容量 - 已经在 StreamByte 中的字节数

// 即剩下可以接受且重组的字节数才是窗口大小

size\_t sz = \_capacity - \_reassembler.stream\_out().buffer\_size();

return sz;

}

除了 segment\_received 复杂一点，ackno 与 window\_size 函数都比较简单，细节可以看看注释。这里额外说明一下：

* ackno 返回确认序号，注意是 WrappingInt32 类型，ackno 必须得 SYN 后才有，并通过 abs\_ackno 与 isn 计算而来。
* window\_size 返回当前接受者的窗口大小，其实就是字节流还能接收多少字节，用容量减去当前的缓冲区大小即可。

segment\_received 是接收者最复杂的函数，需要处理很多边角逻辑，导致代码很多(主要也是因为笔者 C++菜的要死，也懒得重构)，有几个点需要说明一下：

* 第一个包必须携带 SYN，否则直接拒绝；
* 收到 FIN 后需要缓存；
* 得到当前包序号 abs\_seqno 后，需要判断是否超过了窗口；
* stream\_index 与 abs\_seqno 之间的关系转化，通过 first\_syn 来帮助转换；
* \_abs\_ackno 实际为重组确认的当前序号，即 head\_index，但必须包含 SYN 和 FIN。

**TCPSender**

TCPSender 负责从输出流中读取数据并打包发送，主要功能点有：

* 负责接收和缓存 ackno(确认序号)、window\_size(窗口大小);
* 缓存发送但没有被确认的报文，并在超时后重新发送；
* 从输出流中读取数据并包装为报文，然后发送报文。

TCPSender 类定义如下：

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;

//! 当前的重试时间

unsigned int \_current\_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};

//! 发送但未确认的包

std::queue<TCPSegment> \_segments\_in\_flight{};

//! 发送但未确认的字节数

size\_t \_bytes\_in\_flight{0};

//! 滑动窗口大小，默认为 nullopt

std::optional<uint16\_t> \_window\_size{std::nullopt};

//! 确认序列

uint64\_t \_abs\_ackno{0};

//! 已经重试的次数

unsigned int \_consecutive\_retransmissions{0};

//! 定时器

bool \_timer{false};

//! 计时器

unsigned int \_time\_tick{0};

}

* \_isn：初始序号，通过 random 随机生成，并设置到 header 中的 seqno 随报文发送到远端；
* \_segments\_out、\_segments\_in\_flight：报文发送队列和报文已发送未确认队列，sender 只需将发送的报文推入到两个队列中即，其中 \_segments\_out 队列将会由上层的 TCPConnection 处理，而 \_segments\_in\_flight 中的报文需要通过接受到的 ackno 来实时处理，将其中被确认的报文推出队列；
* \_initial\_retransmission\_timeout、\_current\_retransmission\_timeout、\_time\_tick 等：超时相关字段，对于发送但位确认的报文，一旦超时，sender 将重新发送 \_segments\_in\_flight 中的报文，且重试时间是依次递增(\*2)的，一旦将报文推入 \_segments\_in\_flight 那么就开启计时，一旦 \_segments\_in\_flight 为空，即报文均被确认，那么将关闭计时器。

相较于 TCPReceiver，TCPSender 的实现稍微复杂一些，有三个核心函数。

核心函数 fill\_window：负责从输出流中读取数据并打包为 TCP 报文，注意报文大小是有限制的，主要根据窗口大小来实时改变，因此一次可能打包多个报文，如下：

void TCPSender::fill\_window() {

// 根据 window\_size 填充数据到包中

// 首先要发送一个 SYN 包

if (!\_first\_syn) {

\_first\_syn = true;

TCPSegment seg;

seg.header().syn = true;

send\_segment(seg);

return; // 发送 syn 包后等待 ack 才继续发包

}

// Corner Case:

// 再没有 syn 的情况下，不填充

if (!\_first\_syn) {

return; // 其它情况下，未 syn 则不进行任何发送操作

}

// Corner Case:

if (\_fin) {

return; // 都已经 fin 了，还发个鬼啊

}

// \_window\_size 是 0 的时候，sender 将 \_window\_size 当作 1 看待

// 为什么需要看做为 1：因为可以同步 ackno 和 window\_size

size\_t win = \_window\_size.value\_or(0) > 0 ? \_window\_size.value() : 1;

size\_t remain; // 剩余窗口大小

// 当窗口没有满，且无 fin

// \_next\_seqno - \_abs\_ackno 是窗口中待确认的字节数量大小

// 因此还可以发送的字节数 = win - (\_next\_seqno - \_abs\_ackno)

while ((remain = (win - (\_next\_seqno - \_abs\_ackno))) > 0 && !\_fin) {

size\_t sz = min(remain, TCPConfig::MAX\_PAYLOAD\_SIZE);

TCPSegment seg;

string payload = \_stream.read(sz);

seg.payload() = Buffer(move(payload));

// 如果包的大小小于窗口大小，且流已经 eof 了，那么作为 fin(最后一个) 包发送

if (seg.length\_in\_sequence\_space() < win && \_stream.eof()) {

seg.header().fin = true;

\_fin = true;

}

// 注意：fin 包可能没有任何数组，但是如果提前判读包的大小，那么 fin 包

// 在被设置 header 之前就被 return 了，当前的 fin 包就丢失了

if (seg.length\_in\_sequence\_space() == 0) {

return; // 空包直接返回，注意

}

// 每次发送一个包，就会更新 \_next\_seqno，直到无包可发送

send\_segment(seg);

}

}

具体逻辑参考注释，这里额外说明几点：

* 如果是第一次填充，那么必须为 SYN 包，并且打包后直接发送；
* 在没有 SYN 或者已经 FIN 的情况下，直接返回；
* 窗口大小可能为空，必须在接受对端包的情况下才知道对端的接受窗口大小；
* 如果输出流已经关闭，且包长度小于窗口大小，那么作为最后一个包发送，设置为 FIN；
* 如果包长度为 0，那么直接返回，不用发送，空包有另外的处理。

核心函数 ack\_received：接受确认序号(ackno)和窗口大小(window\_size)，处理\_segments\_in\_flight 中的包，将已经确认的报文推出队列，如下：

void TCPSender::ack\_received(const WrappingInt32 ackno, const uint16\_t window\_size) {

// 得到绝对序号 abs\_ackno

uint64\_t abs\_ackno = unwrap(ackno, \_isn, \_next\_seqno);

// Corner Case:

if (abs\_ackno > \_next\_seqno) {

// 太大了，确认不了

return;

}

// 设置 \_window\_size

\_window\_size = make\_optional(window\_size);

// Corner Case:

if (abs\_ackno <= \_abs\_ackno) {

return; // 已经确认过了，不用再确认

}

// 设置 \_abs\_ackno

\_abs\_ackno = abs\_ackno;

// 根据 ackno 移除掉 \_segments\_in\_flight 中的包

while (!\_segments\_in\_flight.empty()) {

TCPSegment front = \_segments\_in\_flight.front();

uint64\_t abs\_seqno = unwrap(front.header().seqno, \_isn, \_next\_seqno);

size\_t len = front.length\_in\_sequence\_space();

// 如果包头部序号 + 长度 <= 确认序号

// 那么表示包被确认了

if (abs\_seqno + len <= \_abs\_ackno) {

\_segments\_in\_flight.pop();

// 计数

\_bytes\_in\_flight -= len;

} else {

break;

}

}

// 收到确认包后，必须重置定时器时间和重试次数

\_current\_retransmission\_timeout = \_initial\_retransmission\_timeout;

\_consecutive\_retransmissions = 0;

// 如果新的包被确认，且还有包未确认，则重新开始定时

if (!\_segments\_in\_flight.empty()) {

\_timer = true;

\_time\_tick = 0;

}

return;

}

说明：

* \_segments\_in\_flight 中的报文如何被确认？很简单，比较报文中的序号、报文长度与确认序号之间的关系，如果报文头部序号 + 长度 <= 确认序号，那么该报文被确认，并且推出队列；
* 另外一旦收到 ackno，那么必须重置超时时间为最初超时时间，且重试次数为 0；
* 所有包确认后，关闭定时器。

核心函数 tick：增加定时器时间，如果发生了超时那么重新发送报文，如下：

void TCPSender::tick(const size\_t ms\_since\_last\_tick) {

// 重新发送最早的包，注意不是所有过期的包

if (!\_timer) {

return;

}

\_time\_tick += ms\_since\_last\_tick;

// 重试一次就 double 重试时间

if (\_time\_tick >= \_current\_retransmission\_timeout && !\_segments\_in\_flight.empty()) {

TCPSegment front = \_segments\_in\_flight.front();

// 重新发送

\_segments\_out.push(front);

// 定时器重置

\_time\_tick = 0;

\_timer = true;

// 超时次数+1

// 注意：When filling window, treat a '0' window size as equal to '1' but don't back off RTO

// 所以如果 window\_size == 0，那么不能对超时时间进行翻倍，只能增加重试次数

\_consecutive\_retransmissions += 1;

// 窗口没有设置过 或者 窗口大小不为 0

if (!\_window\_size.has\_value() || (\_window\_size.has\_value() && \_window\_size.value() != 0)) {

\_current\_retransmission\_timeout \*= 2;

}

}

// 所有数据都被确认了，关闭定时器

if (\_segments\_in\_flight.empty()) {

\_timer = false;

}

}

说明：

* 如果定时器没有开启，那么直接返回，注意定时器在包进入 \_segments\_in\_flight 后才会开启，在队列清空后停止；
* 每次超时重发，只会重发队首的报文，不会重发所有报文；
* 重发后，重试次数 +1，且超时时间 \*=2。

**TCPConnection**

TCPConnection 负责将 TCPSender 和 TCPReceiver 组合起来，维护 TCP 状态机、收、发包等功能。定义如下：

class TCPConnection {

private:

TCPConfig \_cfg;

TCPReceiver \_receiver{\_cfg.recv\_capacity};

TCPSender \_sender{\_cfg.send\_capacity, \_cfg.rt\_timeout, \_cfg.fixed\_isn};

//! outbound queue of segments that the TCPConnection wants sent

std::queue<TCPSegment> \_segments\_out{};

//! Should the TCPConnection stay active (and keep ACKing)

//! for 10 \* \_cfg.rt\_timeout milliseconds after both streams have ended,

//! in case the remote TCPConnection doesn't know we've received its whole stream?

bool \_linger\_after\_streams\_finish{true};

// 上一个包接受距现在时间

size\_t \_time\_since\_last\_segment\_received{0};

//! 是否处于激活状态

bool \_active{true};

}

TCPConnection 的字段比较少：

* \_cfg：TCP 配置，如默认超时时间，最大包长度等；
* \_receiver：接收者；
* \_sender：发送者；
* \_time\_since\_last\_segment\_received：最后报文接受到现在时间间隔；
* \_active：连接是否处于激活状态；
* \_linger\_after\_streams\_finish：是否在数据流结束后等待(涉及到连接关闭，下面会介绍)。

TCPConnection 主要做如下三件事情：

* 接收报文，当 TCP 报文达到时，TCPConnection 的 segment\_received 方法会被调用，并做如下处理：
  + 如果报文头部含有 RST，将输入、输出流设置为错误状态，并且永久关闭连接；
  + 将报文传递给 receiver，即调用 receiver.segment\_received 方法；
  + 如果报文头部中含有 ACK，那么调用 sender.ack\_received 方法；
  + 如果收到的报文的长度 > 0(包含数据、SYN、FIN)，那么必须至少回复一个报文，来告诉对端 ackno 和 window\_size。
* 发送报文：
  + 任何时候，sender 将报文推入到其队列后，TCPConnection 负责将队列中的报文拿出来，设置额外的字段，并放入自己的队列，等待被发送；
  + 在推入报文到队列之前，TCPConnection 必须从 receiver 哪里拿到 ack、ackno 和 window\_size 并填充到报文中。
* 计时：
  + 每当计时(tick)发生时，TCPConnection 负责将消耗时间告诉 sender；
  + 如果报文重试次数超过了 MAX\_RETX\_ATTEMPTS，那么发送 RST 包，并关闭连接；

核心函数 segment\_received 负责接收报文工作：

void TCPConnection::segment\_received(const TCPSegment &seg) {

// 接收包

if (!\_active) {

return;

}

\_time\_since\_last\_segment\_received = 0; // 刷新时间

auto header = seg.header();

// Corner Case:

// ACKs in LISTEN should be ignored

// 如果处于 listen 状态，那么无法接受 ack 包，必须先接受 syn 包

if (header.ack && state() == TCPState::State::LISTEN) {

return;

}

// 如果 header 中有 ack, 那么调用 \_sender.ack\_received

if (header.ack) {

\_sender.ack\_received(header.ackno, header.win);

}

// \_receiver 接收包

\_receiver.segment\_received(seg);

// if the rst (reset) flag is set, sets both the inbound and outbound streams to the error state

// and kills the connection permanently.

// 无论是客户端还是服务端，收到 rst 包后，都应该准备关闭输入、输出

if (header.rst) {

// Corner Case:

// all RSTs should be ignored in LISTEN

// 如果在 listen 状态，那么所有 RST 都应该忽略

if (state() == TCPState::State::LISTEN) {

return;

}

// 关闭连接

unclean\_shutdown();

}

// 是否需要发送空包，空包用于 ack

bool send\_empty = false;

// Corner Case:

// if the incoming segment occupied any sequence numbers,

// the TCPConnection makes sure that at least one segment is sent in reply,

// to reflect an update in the ackno and window size.

// 任何长度大于 0 的包，都应该 ack 回应

if (seg.length\_in\_sequence\_space() > 0) {

send\_empty = true;

}

// Corner Case:

// 如果没有 ackno 确认，即 \_receiver.segment\_received(seg) 失败，则不能发送空包

if (!\_receiver.ackno().has\_value()) {

send\_empty = false;

}

// 尝试去发包回复，注意是尝试，因为如果 \_sender 中无数据发送且也无需发送空包

// 那么 send\_segment 并不会发包

send\_segment(false, false, send\_empty);

}

void TCPConnection::send\_segment(bool syn, bool rst, bool send\_empty) {

// 从 sender 中拿出包，然后再 push\_out

\_sender.fill\_window(); // 填充包

// 如果填充后仍然为空，但必须发送一个空包

if (\_sender.segments\_out().empty() && send\_empty) {

\_sender.send\_empty\_segment();

}

TCPSegment seg;

while (!\_sender.segments\_out().empty()) { // 从 \_sender 的队列中拿出

seg = \_sender.segments\_out().front();

\_sender.segments\_out().pop();

if (syn) {

seg.header().syn = true;

}

if (rst) {

seg.header().rst = true;

}

// 如果 receiver ack 过，那么填充 win, ack, ackno

if (\_receiver.ackno().has\_value()) {

seg.header().ack = true;

seg.header().ackno = \_receiver.ackno().value();

seg.header().win = \_receiver.window\_size();

}

\_segments\_out.push(seg); // 当前发送队列

}

clean\_shutdown();

}

代码上有详细的注释，尤其是一些 Corner Case 需要注意，这里额外说明一下：

* 必须在有 ack 的情况下，才能调用 \_sender.ack\_received；
* 记得调用 \_receiver.segment\_received；
* 任何长度大于 0 的包，都应该至少回应一个包，这个包可能为空包，因此需设置 send\_empty。

发送空包是 sender 提供的函数，如下：

void TCPSender::send\_empty\_segment() {

// 发送空包

TCPSegment seg;

seg.header().seqno = wrap(\_next\_seqno, \_isn);

\_segments\_out.push(seg);

}

那么为什么需要发送空包呢？其实很简单，因为对于远端的包，我们需要及时的回复，但是本端可能没有数据回复，也无需确认(无需加入 \_segments\_in\_flight 队列)，因此需要一个空包(无数据)携带 ackno，window\_size 等字段到达对端同步信息。 注意，TCPConnection 应在任何时候都调用 send\_segment 尝试发送报文，如：

size\_t TCPConnection::write(const string &data) {

// Write data to the outbound byte stream

// 向输出流写数据，即 sender

if (data.size() == 0) {

return 0;

}

size\_t sz = \_sender.stream\_in().write(data);

send\_segment();

return sz;

}

应用程序通过 write 函数向输出流写入数据后，也应该调用 send\_segment 尝试发送报文。 TCPConnection 另一个重要的工作在于计时：

void TCPConnection::tick(const size\_t ms\_since\_last\_tick) {

if (!\_active) {

return;

}

\_time\_since\_last\_segment\_received += ms\_since\_last\_tick;

// 1. tell the TCPSender about the passage of time.

\_sender.tick(ms\_since\_last\_tick);

// 2. abort the connection, and send a reset segment to the peer (an empty segment with the rst flag set),

// if the number of consecutive retransmissions is more than an upper limit TCPConfig::MAX RETX ATTEMPTS.

// 如果超过了重试次数，那么准备关闭连接

if (\_sender.consecutive\_retransmissions() > TCPConfig::MAX\_RETX\_ATTEMPTS) {

// 3. end the connection cleanly if necessary

unclean\_shutdown();

send\_segment(false, true); // 发送 rst 包

}

// 4. 重试发送

// 超时，\_sender 会发送到队列中

// Corner Case:

// 如果处于 listen 状态，那么只应该等待，不应该 send

if (state() != TCPState::State::LISTEN) {

send\_segment();

}

}

每当调用 tick 函数时，增加 \_time\_since\_last\_segment\_received 并且调用 \_sender.tick，然后判断重试次数，如果超过了默认重试次数，那么发送 RST 包，并且关闭连接。 在刚才的代码中，unclean\_shutdown 和 clean\_shutdown 函数多次出现，这两个函数都与连接关闭有直接关系，并且涉及到不同的连接关闭方式。

方式 1：非干净关闭，TCP 连接接收 RST 包，或者被析构、重试次数超过了默认次数，立马关闭输入、输出流，然后设置 active 为 false；

方式 2：干净关闭，没有任何错误产生，双方都全部完成了数据交付和接收，然后设置 active 为 false。

unclean\_shutdown(非干净关闭)是比较简单的，在收到 RST 包、析构、重试失败的时机下调用即可：

void TCPConnection::unclean\_shutdown() {

\_receiver.stream\_out().set\_error();

\_sender.stream\_in().set\_error();

\_active = false;

}

而 clean\_shutdown 是比较麻烦，建立在数据完全交付的情况下，需要满足一下几个条件：

1. 输入流(receiver)已经完全重组且结束；
2. 输出流(sender)已结束，并且已经全部发送到对端；
3. 输出流已经完全被对端确认；
4. ：
   1. 数据流结束后不等待，即 \_linger\_after\_streams\_finish 为 false；
   2. \_time\_since\_last\_segment\_received 已经超过了 10 倍超时时间。

clean\_shutdown 必须同时满足条件 1，3，以及 4 的一个字条件，如下：

void TCPConnection::clean\_shutdown() {

// The inbound stream has been fully assembled and has ended.

if (\_receiver.stream\_out().input\_ended()) {

// The outbound stream has been ended by the local application and fully sent (including

// the fact that it ended, i.e. a segment with fin ) to the remote peer.

if (!\_sender.stream\_in().eof()) {

// If the inbound stream ends before the TCPConnection

// has reached EOF on its outbound stream, this variable needs to be set to false.

\_linger\_after\_streams\_finish = false;

} else if (\_sender.bytes\_in\_flight() == 0) {

// The outbound stream has been fully acknowledged by the remote peer.

if (!\_linger\_after\_streams\_finish || \_time\_since\_last\_segment\_received >= 10 \* \_cfg.rt\_timeout) {

\_active = false;

}

}

}

}

input\_ended 表示输入流结束，bytes\_in\_flight 为 0，表示输出流已经完全被对端接收且确认。 关于 \_linger\_after\_streams\_finish 这个变量需要额外说明一下，它表示流结束后是否需要等待，默认为 true，表示需要等待，因为即使流结束了，但是可能对端没有完全确认，因此需要等待，等待时间为 10 \* \_cfg.rt\_timeout，即子条件 b。

那么如何不等待了？当输入流关闭，而输出流没有结束时，这表示远端已经发送了 FIN 包，所以输入流才能关闭，因此这个状态属于被动关闭，所以无需考虑输出流能否全部交付的问题，因此对端已经不再接受包了，所以彼端可以直接设置 active 为 false。

**Sponge 使用**

到此一个简易版 TCP——Sponge 协议已经实现完毕了，详细代码见参考资料链接。接下来，我们需要去使用一下该协议，从中来看看三次握手是如何发生的。

首先，通过新建 UDPSocket 监听本地 3000 端口：

int main(/\*int argc, char \*\*argv\*/) {

try {

TCPConfig c\_fsm{};

FdAdapterConfig c\_filt{};

Address addr{"127.0.0.1", 3000};

c\_filt.source = addr;

UDPSocket udp\_sock;

udp\_sock.bind(c\_filt.source);

LossyTCPOverUDPSpongeSocket tcp\_socket(LossyTCPOverUDPSocketAdapter(TCPOverUDPSocketAdapter(move(udp\_sock))));

// 开始监听

tcp\_socket.listen\_and\_accept(c\_fsm, c\_filt);

while (!tcp\_socket.eof()) {

const string res = tcp\_socket.read();

cout << "INFO: received from remote: " << res << " \n";

}

tcp\_socket.wait\_until\_closed();

} catch (const exception &e) {

cerr << "Exception: " << e.what() << endl;

return EXIT\_FAILURE;

}

return EXIT\_SUCCESS;

}

UDPSocket 为对 socket API UDP 协议的封装，然后通过 LossyTCPOverUDPSocketAdapter 等适配器将其适配适配到 TCPConnection 上，感兴趣的可以查看源码(非常值得一读)。 绑定本地 3000 端口，然后从 socket 中读取数据。

然后新建客户端连接 3000 端口：

int main(/\*int argc, char \*\*argv\*/) {

try {

TCPConfig c\_fsm{};

FdAdapterConfig c\_filt{};

Address addr{"127.0.0.1", 3000};

c\_filt.destination = addr;

UDPSocket udp\_sock;

LossyTCPOverUDPSpongeSocket tcp\_socket(LossyTCPOverUDPSocketAdapter(TCPOverUDPSocketAdapter(move(udp\_sock))));

// 连接

tcp\_socket.connect(c\_fsm, c\_filt);

// 发送数据

for (size\_t i = 0; i < 10; i++) {

cout << "INFO: wite Hello Pedro! to stream \n";

tcp\_socket.write("Hello Pedro!");

// 休眠 1s

std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));

}

tcp\_socket.wait\_until\_closed();

} catch (const exception &e) {

cerr << "Exception: " << e.what() << endl;

return EXIT\_FAILURE;

}

return EXIT\_SUCCESS;

}

同样的通过 UDPSocket 新建 socket 然后适配到 TCPConnection 中，调用 connect 发起连接，然后向 socket 中定时写入数据。 先运行服务端、再运行客户端。

服务端：

$ ./apps/tcp\_udp\_server

DEBUG: Listening for incoming connection...

DEBUG: [三次握手第二步] 服务端发送 SYN + ACK 包

New connection from 127.0.0.1:39239.

INFO: received from remote: Hello Pedro!

INFO: received from remote: Hello Pedro!

INFO: received from remote: Hello Pedro!

INFO: received from remote: Hello Pedro!

INFO: received from remote: Hello Pedro!

INFO: received from remote: Hello Pedro!

INFO: received from remote: Hello Pedro!

INFO: received from remote: Hello Pedro!

INFO: received from remote: Hello Pedro!

INFO: received from remote: Hello Pedro!

DEBUG: Inbound stream from 127.0.0.1:39239INFO: received from remote:

DEBUG: Waiting for clean shutdown... finished cleanly.

DEBUG: Outbound stream to 127.0.0.1:39239 finished (1 byte still in flight).

DEBUG: Outbound stream to 127.0.0.1:39239 has been fully acknowledged.

DEBUG: TCP connection finished cleanly.

done.

客户端：

$ ./apps/tcp\_udp\_client

DEBUG: Connecting to 127.0.0.1:3000...

DEBUG: [三次握手第一步] 客户端发送 SYN 包

DEBUG: [三次握手第三步] 客户端发送 ACK 包

Successfully connected to 127.0.0.1:3000.

INFO: wite Hello Pedro! to stream

INFO: wite Hello Pedro! to stream

INFO: wite Hello Pedro! to stream

INFO: wite Hello Pedro! to stream

INFO: wite Hello Pedro! to stream

INFO: wite Hello Pedro! to stream

INFO: wite Hello Pedro! to stream

INFO: wite Hello Pedro! to stream

INFO: wite Hello Pedro! to stream

INFO: wite Hello Pedro! to stream

DEBUG: Waiting for clean shutdown... DEBUG: Outbound stream to 127.0.0.1:3000 finished (1 byte still in flight).

DEBUG: Outbound stream to 127.0.0.1:3000 has been fully acknowledged.

DEBUG: Inbound stream from 127.0.0.1:3000 finished cleanly.

DEBUG: Waiting for lingering segments (e.g. retransmissions of FIN) from peer...

DEBUG: TCP connection finished cleanly.

done.

从终端输出中可以看到，双方三次握手的过程，在代码中是这样发生。 首先，客户端调用 connect 函数发出连接，即 SYN 包，这是三次握手的第一步：

void TCPConnection::connect() {

// Initiate a connection by sending a SYN segment

// 客户端发起连接，向服务端发送 SYN 包，三次握手第一步

if (\_first\_syn) {

cerr << "DEBUG: [三次握手第一步] 客户端发送 SYN 包 " << endl;

\_first\_syn = false;

}

send\_segment(true, false);

}

在 connect 函数且没有发生过 SYN 的情况下，就是三次握手的第一步。 三次握手第二步，由服务器发送 SYN + ACK 包：

void TCPConnection::segment\_received(const TCPSegment &seg) {

// 省略代码...

if (header.syn && !header.ack && send\_empty) {

cerr << "DEBUG: [三次握手第二步] 服务端发送 SYN + ACK 包 " << endl;

}

send\_segment(false, false, send\_empty);

}

服务器收到 SYN 包后，判断头部没有 ACK，且发送空包，则向客户端发送 SYN + ACK 包，即三次握手第二步。 三次握手第三步，客户端收到 SYN + ACK 包，发送 ACK 包：

void TCPConnection::segment\_received(const TCPSegment &seg) {

// 省略代码...

if (header.ack) {

if (\_first\_ack && state() == TCPState::State::SYN\_SENT) {

cerr << "DEBUG: [三次握手第三步] 客户端发送 ACK 包 " << endl;

\_first\_ack = false;

}

\_sender.ack\_received(header.ackno, header.win);

}

// 省略代码...

}

客户端在收到第一个 ACK 包，且处于 SYN\_SENT 状态下，接收 ackno，并向服务器发送 ACK 包。至此，三次握手全部完成，双方进入可靠通信阶段，即终端输出的 10 次 Hello Pedro! 。 当客户端数据全部发送完毕，且被确认，客户端会关闭输出流，然后干净地关闭连接，服务端在收到对端关闭后，进入被动关闭状态，等待超时，然后干净地关闭连接(参考终端输出的 DEBUG 日志)。

那么 Sponge 协议的性能怎么样呢？sponge 项目提供了 benchmark 程序，运行结果如下：

./apps/tcp\_benchmark

CPU-limited throughput : 2.35 Gbit/s

CPU-limited throughput with reordering: 1.58 Gbit/s

**总结**

网络协议是一个非常有趣的东西，在互联网普及的今天，基本覆盖了我们生活的方方面面。CS144 是一门好课，将理论知识巧妙和实践结合，在完成 sponge 项目的过程中可以从理论、实践上充分的吸收网络知识，当然 sponge 中还有 IPv4 协议、ARP 协议、路由跳转等多个网络层协议的实现，基本覆盖了网络知识的方方面面，这里就不展开讲了，感兴趣的可以点开参考资料上的连接了解一下。

最后，我们下一个系列再见～

**参考资料**

* [CS 144: Introduction to Computer Networking, Fall 2020](https://cs144.github.io/)
* [深入理解 TCP 协议：从原理到实战](https://juejin.cn/book/6844733788681928712)

最后更新于 2021-09-06 18:11

本文申报

如果觉得我的文章对您有用，请随意赞赏

赏

[3人已赞赏](javascript:void();)

[[](https://km.woa.com/user/doyenli)](https://km.woa.com/user/doyenli)[[](https://km.woa.com/user/johnniexiao)](https://km.woa.com/user/johnniexiao)[](https://km.woa.com/user/maxleisun)

仅供内部学习与交流，未经公司授权切勿外传

标签：[计算机网...](https://km.woa.com/group/tsign/articles?tag_id=34520)(2) [TCP](https://km.woa.com/group/tsign/articles?tag_id=2370)(4) [用户态协...](https://km.woa.com/group/tsign/articles?tag_id=137991)(1)



本文专属二维码，扫一扫还能分享朋友圈

想要微信公众号推广本文章？[点击获取链接](javascript:void(0);)

我顶 (26)

收藏 (104)

* [转载](javascript:void(0);)

* [收录](javascript:void(0);)
* [评论](javascript:void(0);)(9)
* [反馈](javascript:void(0);)

分享到

大家评论

[](https://km.woa.com/user/vvvxu)

[vvvxu](https://km.woa.com/user/vvvxu)

2021-09-02 15:24:58

先收藏再看

 顶 [回复 (1)](javascript:void(0);)

[](https://km.woa.com/user/pedrogao)

[pedrogao](https://km.woa.com/user/pedrogao) (楼主)

2021-09-02 15:37:43

欢迎交流

[回复](javascript:void(0);)

[](https://km.woa.com/user/shawnshuang)

[shawnshuang](https://km.woa.com/user/shawnshuang)

2021-09-03 20:13:39

火钳刘明🙈🙈🙈

 顶 [回复](javascript:void(0);)

[](https://km.woa.com/user/doyenli)

[doyenli](https://km.woa.com/user/doyenli)

2021-09-03 20:15:24



 顶 [回复](javascript:void(0);)

[](https://km.woa.com/user/hazenweng)

[hazenweng](https://km.woa.com/user/hazenweng)

2021-09-05 10:21:00

 排版有点问题，为啥文章中好多处 &amp

 顶 [回复 (1)](javascript:void(0);)

[](https://km.woa.com/user/pedrogao)

[pedrogao](https://km.woa.com/user/pedrogao) (楼主)

2021-09-05 11:03:16

3q，已更正，CV到KM上来的，除了点问题

[回复](javascript:void(0);)

[](https://km.woa.com/user/zekeeliu)

[zekeeliu](https://km.woa.com/user/zekeeliu)

2021-09-06 11:08:15

先m再看

 顶 [回复](javascript:void(0);)

[](https://km.woa.com/user/maxleisun)

[maxleisun](https://km.woa.com/user/maxleisun)

2021-09-07 09:20:14

这也太棒了

 顶 [回复 (1)](javascript:void(0);)

[](https://km.woa.com/user/pedrogao)

[pedrogao](https://km.woa.com/user/pedrogao) (楼主)

2021-09-07 09:26:40

谢谢，多多交流

[回复](javascript:void(0);)

[](https://km.woa.com/user/rickyrqzhao)



[切换到更多功能](javascript:void(0);)

关于作者

[](https://km.woa.com/user/pedrogao)

[pedrogao(高佩东)](https://km.woa.com/user/pedrogao)

CSIG\企业中台产品部\后台研发组员工

作者文章

* [谈谈用户态 TCP 协议实现](https://km.woa.com/posts/show/519957?kmref=author_post)
* [从 Hello World 来看看 Go 的运行流程](https://km.woa.com/posts/show/515314?kmref=author_post)
* [Go 汇编简单入门](https://km.woa.com/posts/show/515137?kmref=author_post)

Copyright©1998-2021 Tencent Inc. All Rights Reserved

腾讯公司研发管理部 版权所有

[广告申请](https://km.woa.com/chartlets/km/) [反馈问题](javascript:void(0);)

[575/612/401 ms]