Lab Checkpoint 2: the TCP receiver

Due: Sunday, October 12, 11:59 p.m.

Collaboration Policy: Same as checkpoint 0. Please do not look at other students' or chatbots' code or solutions to past versions of these assignments. Please fully disclose any collaborators or any gray areas in your writeup—disclosure is the best policy.

0 Overview

Suggestion: read the whole lab document before implementing.

In Checkpoint 0, you implemented the abstraction of a *flow-controlled byte stream* (`ByteStream`). And in Checkpoint 1, you created a `Reassembler` that accepts a sequence of substrings, all excerpted from the same byte stream, and reassembles them back into the original stream.

These modules will prove useful in your TCP implementation, but nothing in them was specific to the details of the Transmission Control Protocol. That changes now. In Checkpoint 2, you will implement the `TCPReceiver`, the part of a TCP implementation that handles the incoming byte stream.

The `TCPReceiver` receives messages from the peer's sender (via the `receive()` method) and turns them into calls to a `Reassembler`, which eventually writes to the incoming `ByteStream`. Applications read from this `ByteStream`, just as you did in Lab 0 by reading from the `TCPSocket`.

Meanwhile, the `TCPReceiver` also generates messages that go back to the peer's sender, via the `send()` method. These "receiver messages" are responsible for telling the sender:

1. the index of the "first unassembled" byte, which is called the "acknowledgment number" or "**ackno**." This is the first byte that the receiver needs from the sender.
2. the available capacity in the output ByteStream. This is called the "**window size**".

Together, the **ackno** and **window size** describe describes the receiver's **window**: a **range of indexes** that the TCP sender is allowed to send. Using the window, the receiver can control the flow of incoming data, making the sender limit how much it sends until the receiver is ready for more. We sometimes refer to the ackno as the "left edge" of the window (smallest index the `TCPReceiver` is interested in), and the ackno + window size as the "right edge" (just beyond the largest index the `TCPReceiver` is interested in).

You've already done most of the algorithmic work involved in implementing the `TCPReceiver` when you wrote the `Reassembler` and `ByteStream`; this lab is about wiring those general classes up to the details of TCP. The hardest part will involve thinking about how TCP will represent each byte's place in the stream—known as a "sequence number."

实验检查点2：TCP接收器

截止日期：2025年10月12日星期日23:59

协作政策：与检查点0相同。请不要查看其他学生或聊天机器人的代码或这些作业旧版本的解决方案。请在您的报告中充分披露任何合作者或任何灰色地带——披露是最好的政策。

0 概述

建议：在实现前，先阅读整个实验文档。

在检查点0中，你实现了流控制字节流的抽象(`ByteStream`)。而在检查点1中，你创建了一个 `Reassembler`，它接受一系列从同一字节流中提取的子字符串，并将它们重新组装成原始流。

这些模块在你TCP实现中会很有用，但它们中的任何内容都不是传输控制协议细节特定的。现在情况变了。在检查点2中，你将实现 `TCPReceiver`，即TCP实现中处理传入字节流的部分。

该 `TCPReceiver` 通过 `receive()` 方法从对端的发送方接收消息，并将它们转换为对 `Reassembler`的调用，后者最终写入传入的 `ByteStream`。应用程序从这个 `ByteStream`读取数据，就像你在实验0中通过从`TCPSocket`读取数据一样。

与此同时，该 `TCPReceiver` 还通过 `send()` 方法生成消息发送回对端的发送方。这些“接收器消息”负责告诉发送方：

1. “首次未组装”字节的索引，称为“确认号”或“ackno.”。这是接收方需要从发送方获取的第一个字节。
2. 输出字节流中的可用容量。这称为“窗口大小”。

两者结合，ackno.和窗口大小描述了接收方的窗口：TCP发送方被允许发送的索引范围。使用窗口，接收方可以控制传入数据的流量，使发送方限制发送量，直到接收方准备好接收更多数据。我们有时将ackno.称为窗口的“左边缘”（ `TCPReceiver` 最感兴趣的索引），将ackno. + 窗口大小称为“右边缘”（ `TCPReceiver` 最感兴趣的索引之后）。

在编写 `Reassembler` 和 `ByteStream` 时，你已经完成了实现 `TCPReceiver` 的大部分算法工作；这个实验是关于将那些通用类与 TCP 的细节连接起来的。最难的部分将涉及思考 TCP 如何表示流中每个字节的位置——这被称为“序列号”。

1 Getting started

Your implementation of a `TCPReceiver` will use the same Minnow library that you used in Checkpoints 0 and 1, with additional classes and tests. To get started:

1. Make sure you have committed all your solutions to Checkpoint 1. Please don't modify any files outside the top level of the `src` directory, or `webget.cc`. You may have trouble merging the Checkpoint 1 starter code otherwise.
2. While inside the repository for the lab assignments, run `git fetch --all` to retrieve the most recent version of the lab assignment.
3. Download the starter code for Checkpoint 2 by running `git merge origin/check2-startercode`. (If you have renamed the "origin" remote to be something else, you might need to use a different name here, e.g. `git merge upstream/check2-startercode`.)
4. Make sure your build system is properly set up: `cmake -S . -B build`
   - **Note for arm64 (UTM) Mac users:** In the past, the g++ "sanitizers" (bug checkers) ran *very* slow on arm64. Minnow uses these to run the tests. If you are on an arm64 Mac, and if you are having trouble, you can configure `cmake` to use a different compiler:

     `cmake -S . -B build -DCMAKE_CXX_COMPILER=clang++`
5. Compile the source code: `cmake --build build`
6. Open and start editing the `writeups/check2.md` file. This is the template for your lab writeup and will be included in your submission.

2 Checkpoint 2: The TCP Receiver

TCP is a protocol that reliably conveys a pair of flow-controlled byte streams (one in each direction) over unreliable datagrams. Two parties, or "peers," participate in the TCP connection, and *each peer* acts as both "sender" (of its own outgoing byte stream) and "receiver" (of an incoming byte stream) at the same time.

This week, you'll implement the "receiver" part of TCP, responsible for receiving messages from the sender, reassembling the byte stream (including its ending, when that occurs), and determining that messages that should be sent back to the sender for acknowledgment and flow control.

> ⋆*Why am I doing this?* These signals are crucial to TCP's ability to provide the service of a flow-controlled, reliable byte stream over an unreliable datagram network. In TCP, **acknowledgment** means, "What's the index of the *next* byte that the receiver needs so it can reassemble more of the `ByteStream`?" This tells the sender what bytes it needs to send or resend. **Flow control** means, "What range of indices is the receiver interested and willing to receive?" (a function of its available capacity). This tells the sender how much it's *allowed* to send.

1 开始使用

你的 `TCPReceiver` 实现将使用你在检查点 0 和 1 中使用的相同 Minnow库，并添加额外的类和测试。开始操作：

1. 确保你已经将所有解决方案提交到检查点1。请不要修改 `src` 目录顶层以外的任何文件，或者 `webget.cc`。否则，你可能会在合并检查点1的启动代码时遇到问题。
2. 在实验任务的存储库内，运行 `git fetch --all` 来获取实验任务最新的版本。
3. 通过运行 `git merge origin/check2-startercode` 下载检查点2的启动代码。（如果你将“origin”远程仓库重命名为其他名称，这里可能需要使用不同的名称，例如 `git merge upstream/check2-startercode`。）
4. 确保你的构建系统已正确设置： `cmake -S . -B build`
   - arm64（UTM）Mac用户注意：过去，g++ “检查器”（错误检查器）在arm64上的运行速度非常慢。Minnow使用这些来运行测试。如果你使用的是arm64 Mac，并且遇到问题，你可以配置 `cmake` 以使用不同的编译器：

     `cmake -S . -B build -DCMAKE_CXX_COMPILER=clang++`
5. 编译源代码： `cmake --build build`
6. 打开并开始编辑 `writeups/check2.md` 文件。这是你的实验报告模板，并将包含在你的提交中。

2 检查点2：TCP接收器

TCP是一种协议，能够在不可靠的数据报上可靠地传输一对流量控制的字节流（每个方向一个）。两个参与方，或称为“对等端”，参与TCP连接，并且每个对等端同时充当“发送者”（发送自己的字节流）和“接收者”（接收传入的字节流）。

本周，你将实现TCP的“接收者”部分，负责接收来自发送者的消息，重新组装字节流（包括其结束，当发生时），并确定哪些消息需要发送回发送者以进行确认和流量控制。

> ⋆我为什么要做这个？这些信号对于 TCP 提供流控、可靠的字节流服务至关重要。在 TCP 中，确认意味着“接收者需要下一个字节的索引是多少，以便它可以重新组装更多的 `ByteStream`？”这告诉发送者它需要发送或重发的字节。流控意味着“接收者感兴趣且愿意接收的索引范围是多少？”（这取决于它的可用容量）。这告诉发送者它被允许发送多少数据。

2.1 Translating between 64-bit indexes and 32-bit seqnos

As a warmup, we'll need to implement TCP's way of representing indexes. Last week you created a `Reassembler` that reassembles substrings where each individual byte has a 64-bit **stream index**, with the first byte in the stream always having index zero. A 64-bit index is big enough that we can treat it as **never overflowing**.[1] In the TCP headers, however, space is precious, and each byte's index in the stream is represented not with a 64-bit index but with a 32-bit "sequence number," or "seqno." This adds three complexities:

1. **Your implementation needs to plan for 32-bit integers to wrap around.** Streams in TCP can be arbitrarily long—there's no limit to the length of a ByteStream that can be sent over TCP. But 2^{32} bytes is only 4 GiB, which is not so big. Once a 32-bit sequence number counts up to $2^{32} - 1$, the next byte in the stream will have the sequence number zero.

2. **TCP sequence numbers start at a random value:** To improve robustness and avoid getting confused by old segments belonging to earlier connections between the same endpoints, TCP tries to make sure sequence numbers can't be guessed and are unlikely to repeat. So the sequence numbers for a stream don't start at zero. The first sequence number in the stream is a *random 32-bit number* called the Initial Sequence Number (ISN). This is the sequence number that represents the "zero point" or the SYN (beginning of stream). The rest of the sequence numbers behave normally after that: the first byte of data will have the sequence number of the ISN+1 (mod 2^{32}), the second byte will have the ISN+2 (mod 2^{32}), etc.

3. **The logical beginning and ending each occupy one sequence number:** In addition to ensuring the receipt of all bytes of data, TCP makes sure that the beginning and ending of the stream are received reliably. Thus, in TCP the SYN (beginning-of-stream) and FIN (end-of-stream) control flags are assigned sequence numbers. Each of these occupies *one* sequence number. (The sequence number occupied by the SYN flag is the ISN.) Each byte of data in the stream also occupies one sequence number. Keep in mind that SYN and FIN aren't part of the stream itself and aren't "bytes"—they represent the beginning and ending of the byte stream itself.

These sequence numbers (**seqnos**) are transmitted in the header of each TCP segment. (And, again, there are two streams—one in each direction. Each stream has separate sequence numbers and a different random ISN.) It's also sometimes helpful to talk about the concept of an **"absolute sequence number"** (which always starts at zero and doesn't wrap), and about a **"stream index"** (what you've already been using with your `Reassembler`: an index for each byte in the stream, starting at zero).

To make these distinctions concrete, consider the byte stream containing just the three-letter string `'cat'`. If the SYN happened to have seqno $2^{32} - 2$, then the seqnos, absolute seqnos, and stream indices of each byte are:

[1] Transmitting at 100 gigabits/sec, it would take almost 50 years to reach 2^{64} bytes. By contrast, it takes only a third of a second to reach 2^{32} bytes.

2.1 在64位索引和32位序列号之间进行转换

作为热身，我们需要实现 TCP 表示索引的方式。上周你创建了一个 `Reassembler`，用于重新组装子字符串，其中每个字节都有一个 64 位流索引，流中的第一个字节索引始终为零。64 位索引足够大，我们可以将其视为永远不会溢出。[1] 然而，在 TCP 头部中，空间非常宝贵，流中每个字节的索引不是用 64 位索引表示，而是用一个 32 位的“序列号”，或“seqno”表示。这增加了三个复杂性：

1. 你的实现需要考虑32位整数会溢出。TCP中的流可以是任意长的——没有限制可以在TCP上发送的字节流的大小。但 2^{32} 字节只有4 GiB，这并不算大。一旦一个32位序列号计数到 $2^{32} - 1$，流中的下一个字节将具有序列号零。

2. TCP序列号从随机值开始：为了提高鲁棒性并避免因旧段属于相同端点之间早期连接而混淆，TCP试图确保序列号无法被猜测且不太可能重复。因此，流的序列号不会从零开始。流中的第一个序列号是一个随机的32位数，称为初始序列号（ISN）。这是代表“零点”或SYN（流开始）的序列号。从那时起，其余的序列号正常工作：第一个数据字节将具有ISN+1 （mod 2^{32}）的序列号，第二个字节将具有ISN+2 （mod 2^{32}），等等。

3. 逻辑起始和结束各占用一个序列号：除了确保接收所有数据字节外，TCP 还确保流的开头和结尾能够可靠接收。因此，在 TCP 中，流的开始（SYN）和结束（FIN）控制标志被分配序列号。这些标志各占用一个序列号。（SYN 标志占用的序列号是 ISN。）流中的每个数据字节也占用一个序列号。请注意，SYN 和 FIN 不是流本身的一部分，也不是“字节”——它们代表字节流的开始和结束。

这些序列号（序列号）通过每个 TCP 段的头部传输。（而且，再次强调，存在两个流——每个方向一个。每个流有独立的序列号和不同的随机 ISN。）有时讨论“绝对序列号”（始终从零开始且不回绕）和“流索引”（您已经使用 `Reassembler` 的那种：流中每个字节的索引，从零开始）的概念也很有帮助。

为了让这些区别更具体，考虑一个只包含三个字母的字符串 `'cat'`的字节流。如果 SYN 发生时恰好有 seqno $2^{32} - 2$，那么每个字节的 seqno、绝对序列号和流索引如下：

[1] 以每秒100吉比特的速度传输，到达 2^{64} 字节几乎需要50年。相比之下，到达 2^{32} 字节仅需三分之一秒。

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	

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

Sequence Numbers	**Absolute Sequence Numbers**	**Stream Indices**
• Start at the ISN	• Start at 0	• Start at 0
• Include SYN/FIN	• Include SYN/FIN	• Omit SYN/FIN
• 32 bits, wrapping	• 64 bits, non-wrapping	• 64 bits, non-wrapping
• "seqno"	• "absolute seqno"	• "stream index"

Converting between absolute sequence numbers and stream indices is easy enough—just add or subtract one. Unfortunately, converting between sequence numbers and absolute sequence numbers is a bit harder, and confusing the two can produce tricky bugs. To prevent these bugs systematically, we'll represent sequence numbers with a custom type: `Wrap32`, and write the conversions between it and absolute sequence numbers (represented with `uint64_t`). `Wrap32` is an example of a *wrapper type*: a type that contains an inner type (in this case `uint32_t`) but provides a different set of functions/operators.

We've defined the type for you and provided some helper functions, but you'll implement the conversions in `wrapping_integers.cc`:

1. `static Wrap32 Wrap32::wrap( uint64_t n, Wrap32 zero_point )`

   Convert absolute seqno → seqno. Given an absolute sequence number (n) and an Initial Sequence Number (`zero_point`), produce the (relative) sequence number for n.

2. `uint64_t unwrap( Wrap32 zero_point, uint64_t checkpoint ) const`

   Convert seqno → absolute seqno. Given a sequence number (the `Wrap32`), the Initial Sequence Number (`zero_point`), and an absolute *checkpoint* sequence number, find the corresponding absolute sequence number that is **closest to the checkpoint.**

   Note: A **checkpoint** is required because any given seqno corresponds to *many* absolute seqnos. E.g. with an ISN of zero, the seqno "17" corresponds to the absolute seqno of 17, but also $2^{32}+17$, or $2^{33}+17$, or $2^{33}+2^{32}+17$, or $2^{34}+17$, or $2^{34}+2^{32}+17$, etc. The checkpoint helps resolve the ambiguity: it's an absolute seqno that the user of this class knows is "in the ballpark" of the correct answer. In your TCP implementation, you'll use the first unassembled index as the checkpoint.

   Hint: *The cleanest/easiest implementation will use the helper functions provided in* `wrapping_integers.hh`. *The wrap/unwrap operations should preserve offsets—two seqnos that differ by 17 will correspond to two absolute seqnos that also differ by 17.*

   Hint #2: *We're expecting one line of code for wrap, and less than 10 lines of code for unwrap. If you find yourself implementing a lot more than this, it might be wise to step back and try to think of a different strategy.*

元素	SYN	c	a	t	FIN
序列号	$2^{32}-2$	$2^{32}-1$	0	1	2
绝对序列号	0	1	2	3	4
流索引		0	1	2	

该图显示了 TCP 中涉及的三种不同索引类型：

序列号	**绝对序列号 StreamIndices**	
• 从 ISN 开始	• 从 0 开始	• 从 0 开始
• 包含 SYN/FIN	• 包含SYN/FIN	• 省略SYN/FIN
• 32位，回绕	• 64位，非回绕	• 64位，非回绕
• "序列号"	• "绝对序列号"	• "流索引"

在绝对序列号和流索引之间转换很简单——只需加一或减一。不幸的是，在序列号和绝对序列号之间转换会稍微困难一些，混淆两者可能导致棘手的错误。为了系统性地防止这些错误，我们将使用自定义类型 `Wrap32` 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。`Wrap32` 是一个包装类型的例子：一种包含内部类型（在本例中为 `uint32 t`）但提供不同函数/操作符的类型。

我们已经为您定义了类型并提供了辅助函数，但您将在 `wrapping integers.cc` 中实现转换：

1. `static Wrap32 Wrap32::wrap( uint64 t n, Wrap32 zero point )`

   将绝对序列号 → seqno 转换。给定一个绝对序列号 (n) 和一个初始序列号 (`zero point`)，用于生成 n 的（相对）序列号。

2. `uint64 t unwrap( Wrap32 zero point, uint64 t checkpoint ) const`

   将 seqno → 转换为绝对序列号。给定一个序列号（即 `Wrap32`）、初始序列号 (`zero point`) 以及一个绝对检查点序列号，找出最接近该检查点的绝对序列号。

   注意：需要检查点是因为任何给定的 seqno 都可能对应多个绝对序列号。例如，当 ISN 为零时，seqno "17" 对应的绝对序列号是 17，但也可能是 $2^{32}+17$、$2^{33}+17$、$2^{33}+2^{32}+17$、$2^{34}+17$ 或 $2^{34}+2^{32}+17$ 等。检查点有助于消除这种歧义：它是一个用户知道属于正确答案"大致范围"内的绝对序列号。在你的 TCP 实现中，你会使用第一个未组装的索引作为检查点。

   提示：最干净/最简单的实现将使用`wrapping integers.hh`中提供的辅助函数。封装/解封装操作应保持偏移量——两个相差*17*的序列号将对应两个相差*17*的绝对序列号。

   提示 *#2*：我们期望 *wrap* 只需要一行代码，而 *unwrap* 不超过 *10* 行代码。如果你发现自己实现的代码量远超这个范围，那么最好停下来，尝试思考不同的策略。

You can test your implementation by running the tests: `cmake --build build --target check2`. (Reminder: Mac arm64 users may find they want to configure cmake to use the "clang++" compiler—see above.)

2.2 Implementing the TCP receiver

Congratulations on getting the wrapping and unwrapping logic right! We'll shake your hand (or, post-COVID, elbow-bump) if this victory happens at the lab session. In the rest of this lab, you'll be implementing the `TCPReceiver`. It will (1) receive messages from its peer's sender and reassemble the `ByteStream` using a `Reassembler`, and (2) send messages back to the peer's sender that contain the acknowledgment number (ackno) and window size. We're expecting this to take **about 15 lines of code** in total.

First, let's review the format of a TCP "sender message," which contains the information about the `ByteStream`. These messages are sent *from* a `TCPSender` to its peer's `TCPReceiver`:

```
/*
 * The TCPSenderMessage structure contains five fields (minnow/util/tcp_sender_message.hh):
 *
 * 1) The sequence number (seqno) of the beginning of the segment. If the SYN flag is set,
 *    this is the sequence number of the SYN flag. Otherwise, it's the sequence number of
 *    the beginning of the payload.
 *
 * 2) The SYN flag. If set, this segment is the beginning of the byte stream, and the seqno field
 *    contains the Initial Sequence Number (ISN) -- the zero point.
 *
 * 3) The payload: a substring (possibly empty) of the byte stream.
 *
 * 4) The FIN flag. If set, the payload represents the ending of the byte stream.
 *
 * 5) The RST (reset) flag. If set, the stream has suffered an error and the connection
 *    should be aborted.
 */

struct TCPSenderMessage
{
  Wrap32 seqno { 0 };

  bool SYN {};
  std::string payload {};
  bool FIN {};

  bool RST {};

  // How many sequence numbers does this segment use?
  size_t sequence_length() const { return SYN + payload.size() + FIN; }
};
```


你可以通过运行测试来测试你的实现：`cmake --build build --target check2`。(提醒：Mac arm64 用户可能会发现需要配置 cmake 使用"clang++"编译器——见上文。)

2.2 实现 TCP 接收器

恭喜你正确实现了封装和解封装逻辑！如果在实验课上取得这个胜利，我们会握手（或，在 post-COVID 后，碰肘）向你表示祝贺。在本实验的其余部分，你将实现 `TCPReceiver`。它将 (1) 从对端的发送器接收消息，并使用 `Reassembler` 重组 `ByteStream`，以及 (2) 向对端的发送器发送包含确认号（ackno）和窗口大小的消息。我们预计总共需要大约 15 行代码。

首先，让我们回顾一下 TCP "发送器消息"的格式，其中包含关于 `ByteStream` 的信息。这些消息是从 `TCPSender` 发送到对端的 `TCPReceiver` 的：

```
/*
 * The TCPSenderMessage structure contains five fields (minnow/util/tcp_sender_message.hh):
 *
 * 1) The sequence number (seqno) of the beginning of the segment. If the SYN flag is set,
 *    this is the sequence number of the SYN flag. Otherwise, it's the sequence number of
 *    the beginning of the payload.
 *
 * 2) The SYN flag. If set, this segment is the beginning of the byte stream, and the seqno field
 *    contains the Initial Sequence Number (ISN) -- the zero point.
 *
 * 3) The payload: a substring (possibly empty) of the byte stream.
 *
 * 4) The FIN flag. If set, the payload represents the ending of the byte stream.
 *
 * 5) The RST (reset) flag. If set, the stream has suffered an error and the connection
 *    should be aborted.
 */

struct TCPSenderMessage
{
  Wrap32 序列号 { 0 };

  bool SYN {};
  std::string payload {};
  bool FIN {};

  bool RST {};

  // How many sequence numbers does this segment use?
  size_t sequence_length() const { return SYN + payload.size() + FIN; }
};
```

The `TCPReceiver` generates its own messages back to the peer's `TCPSender`:

```
/*
 * The TCPReceiverMessage structure contains three fields (minnow/util/tcp_receiver_message.hh):
 *
 * 1) The acknowledgment number (ackno): the *next* sequence number needed by the TCP Receiver.
 *    This is an optional field that is empty if the TCPReceiver hasn't yet received the
 *    Initial Sequence Number.
 *
 * 2) The window size. This is the number of sequence numbers that the TCP receiver is interested
 *    to receive, starting from the ackno if present. The maximum value is 65,535 (UINT16_MAX from
 *    the <cstdint> header).
 *
 * 3) The RST (reset) flag. If set, the stream has suffered an error and the connection
 *    should be aborted.
 */

struct TCPReceiverMessage
{
  std::optional<Wrap32> ackno {};
  uint16_t window_size {};
  bool RST {};
};
```

Your `TCPReceiver`'s job is to receive one of these kinds of messages and send the other:

```
class TCPReceiver
{
public:
  // Construct with given Reassembler
  explicit TCPReceiver( Reassembler&& reassembler ) : reassembler_( std::move( reassembler ) )

  // The TCPReceiver receives TCPSenderMessages from the peer's TCPSender.
  void receive( TCPSenderMessage message );

  // The TCPReceiver sends TCPReceiverMessages to the peer's TCPSender.
  TCPReceiverMessage send() const;

  // Access the output (only Reader is accessible non-const)
  const Reassembler& reassembler() const { return reassembler_; }
  Reader& reader() { return reassembler_.reader(); }
  const Reader& reader() const { return reassembler_.reader(); }
  const Writer& writer() const { return reassembler_.writer(); }

private:
  Reassembler reassembler_;
};
```


该 `TCPReceiver` 向对端的 `TCPSender` 生成自己的消息：

```
/*
 * The TCPReceiverMessage structure contains three fields (minnow/util/tcp_receiver_message.hh):
 *
 * 1) The acknowledgment number (ackno): the *next* sequence number needed by the TCP Receiver.
 *    This is an optional field that is empty if the TCPReceiver hasn't yet received the
 *    Initial Sequence Number.
 *
 * 2) The window size. This is the number of sequence numbers that the TCP receiver is interested
 *    to receive, starting from the ackno if present. The maximum value is 65,535 (UINT16_MAX from
 *    the <cstdint> header).
 *
 * 3) The RST (reset) flag. If set, the stream has suffered an error and the connection
 *    should be aborted.
 */

struct TCPReceiverMessage
{
  std::optional<Wrap32> ackno {};
  uint16_t window_size {};
  bool RST {};
};
```

您的 `TCPReceiver` 的任务是接收其中一种消息类型并发送另一种：

```
class TCPReceiver
{
public:
  // Construct with given Reassembler
  explicit TCPReceiver( Reassembler&& reassembler ) : reassembler_( std::move( reassembler ) )

  // The TCPReceiver receives TCPSenderMessages from the peer's TCPSender.
  void receive( TCPSenderMessage message );

  // The TCPReceiver sends TCPReceiverMessages to the peer's TCPSender.
  TCPReceiverMessage send() const;

  // Access the output (only Reader is accessible non-const)
  const Reassembler& reassembler() const { return reassembler_; }
  Reader& reader() { return reassembler_.reader(); }
  const Reader& reader() const { return reassembler_.reader(); }
  const Writer& writer() const { return reassembler_.writer(); }

private:
  Reassembler reassembler_;
};
```

2.2.1 receive()

This is method will be called each time a new segment is received from the peer's sender. This method needs to:

- **Set the Initial Sequence Number if necessary.** The sequence number of the first-arriving segment that has the SYN flag set is the *initial sequence number*. You'll want to keep track of that in order to keep converting between 32-bit wrapped seqnos/acknos and their absolute equivalents. (Note that the SYN flag is just *one* flag in the header. The same message could also carry data or have the FIN flag set.)
- **Push any data to the `Reassembler`.** If the `FIN` flag is set in a `TCPSegment`'s header, that means that the last byte of the payload is the last byte of the entire stream. Remember that the `Reassembler` expects stream indexes starting at zero; you will have to unwrap the seqnos to produce these.
- **Not duplicate any state or functionality that something else already provides.** You have already built the `ByteStream`, `Reassembler`, and `Wrap32` modules, and they do some very useful things! The `TCPReceiver` shouldn't keep track of any state (member variables), and shouldn't duplicate any functionality, that these modules already handle.

3 Development and debugging advice

1. Implement the `TCPReceiver`'s public interface (and any private methods or functions you'd like) in the file `tcp_receiver.cc`. You may add any private members you like to the `TCPReceiver` class in `tcp_receiver.hh`.
2. You can test your code with `cmake --build build --target check2`.
3. Please re-read the section on "using Git" in the Lab 0 document, and remember to keep the code in the Git repository it was distributed in on the `main` branch. Make small commits, using good commit messages that identify what changed and why.
4. Please work to make your code readable to the CA who will be grading it for style. Use reasonable and clear naming conventions for variables. Use comments to explain complex or subtle pieces of code. Use "defensive programming"—explicitly check preconditions of functions or invariants, and throw an exception if anything is ever wrong. Use modularity in your design—identify common abstractions and behaviors and factor them out when possible. Blocks of repeated code and enormous functions will make your code harder to follow.
5. Please also keep to the "Modern C++" style described in the Checkpoint 0 document. The cppreference website (https://en.cppreference.com) is a great resource, although you won't need any sophisticated features of C++ to do these labs.

2.2.1 receive()

这个方法会在每次从对端的发送方接收到新片段时被调用。这个方法需要：

- 如果需要，设置初始序列号。具有SYN标志的第一个到达片段的序列号是初始序列号。您需要跟踪这个，以便在32位环绕序列号/确认号和它们的绝对值之间进行转换。（注意，SYN标志只是头部中的一个标志。相同的信息也可能携带数据或设置FIN标志。）
- 将任何数据推送到 `Reassembler`。如果一个 `TCPSegment`的头部中设置了 `FIN` 标志，这意味着有效载荷的最后字节是整个流的最后字节。请记住， `Reassembler` 期望流索引从零开始；您将不得不解包序列号来生成这些。
- 不要重复任何其他东西已经提供的任何状态或功能。你已经构建了 `ByteStream`、`Reassembler`和 `Wrap32` 模块，它们做了一些非常有用的事情！ `TCPReceiver`不应该跟踪任何状态（成员变量），也不应该重复任何这些模块已经处理的功能。

3 开发和调试建议

1. 在文件 `tcp receiver.cc`中实现 `TCPReceiver`的公共接口（以及任何你想添加的私有方法或函数）。你可以将任何你喜欢的私有成员添加到 `TCPReceiver` 类中的 `tcp receiver.hh`。
2. 你可以使用 `cmake --build build --target check2` 测试你的代码。
3. 请重新阅读实验室0文档中关于“使用Git”的部分，并记住要在 `main` 分支上保持代码在分发时所在的Git存储库中。进行小的提交，使用好的提交信息来标识更改的内容和原因。
4. 请努力让你的代码对将为你评分风格的CA来说是可读的。为变量使用合理和清晰的命名约定。使用注释来解释复杂的或微妙的代码部分。使用“防御性编程”——明确检查函数的前置条件或不变量，如果有任何错误就抛出异常。在你的设计中使用模块化——识别常见的抽象和行为，并在可能的情况下将它们提取出来。重复的代码块和巨大的函数会使你的代码更难理解。
5. 请同时遵循检查点0文档中描述的“现代C++”风格。cppreference网站（https://en.cppreference.com）是一个很好的资源，不过你完成这些实验并不需要用到C++的任何高级特性。

4 Submit

1. In your submission, please only make changes to the `.hh` and `.cc` files in the `src` directory. Within these files, please feel free to add private members as necessary, but please don't change the *public* interface of any of the classes.

2. Before handing in any assignment, please run these in order:

   (a) Make sure you have committed all of your changes to the Git repository. You can run `git status` to make sure there are no outstanding changes. Remember: make small commits as you code.

   (b) `cmake --build build --target format` (to normalize the coding style)

   (c) `cmake --build build --target check2` (to make sure the automated tests pass)

   (d) Optional: `cmake --build build --target tidy` (suggests improvements to follow good C++ programming practices)

3. Write a report in `writeups/check2.md`. This file should be a roughly 20-to-50-line document with no more than 80 characters per line to make it easier to read. The report should contain the following sections:

   (a) **Program Structure and Design.** Describe the high-level structure and design choices embodied in your code. You do not need to discuss in detail what you inherited from the starter code. Use this as an opportunity to highlight important design aspects and provide greater detail on those areas for your grading TA to understand. You are strongly encouraged to make this writeup as readable as possible by using subheadings and outlines. Please do not simply translate your program into an paragraph of English.

   (b) **Alternative design choices** that you considered or ideally evaluated in terms of their performance, difficulty to write (e.g., hours required to produce a bug-free implementation), difficulty to read (e.g., lines of code and their degree of subtlety or nonobvious correctness), and any other dimensions you think are interesting for the reader (or for your own past self before you did this assignment). Include any measurements if applicable.

   (c) **Implementation Challenges.** Describe the parts of code that you found most troublesome and explain why. Reflect on how you overcame those challenges and what helped you finally understand the concept that was giving you trouble. How did you attempt to ensure that your code maintained your assumptions, invariants, and preconditions, and in what ways did you find this easy or difficult? How did you debug and test your code?

   (d) **Remaining Bugs.** Point out and explain as best you can any bugs (or unhandled edge cases) that remain in the code.

4 提交

1. 在你的提交中，请仅修改 `.hh` 和 `.cc` 文件，这些文件位于 `src` 目录中。在这些文件中，你可以根据需要添加私有成员，但请勿更改任何类的公共接口。

2. 提交任何作业前，请按顺序运行这些：

   (a) 确保你已经将所有更改提交到 Git 仓库。你可以运行 `git status` 来确保没有未完成的更改。记住：边编写代码边进行小规模提交。

   (b) `cmake --build build --target format`（用于规范化编码风格）

   (c) `cmake --build build --target check2`（确保自动化测试通过）

   (d) 可选：`cmake --build build --target tidy`（建议改进遵循良好的 C++编程实践）

3. 在 `writeups/check2.md`中撰写报告。该文件应是一个约20至50行的文档，每行不超过80个字符，以便于阅读。报告应包含以下部分：

   (a) 程序结构和设计。描述代码体现的高级结构和设计选择。你不需要详细讨论从启动代码继承的内容。利用这个机会突出重要的设计方面，并为你的评分助教提供更多细节以理解。强烈建议你通过使用子标题和提纲使这份写稿尽可能易读。请不要简单地将你的程序翻译成一段英文。

   (b) 你考虑过的替代设计选择，或者理想情况下根据性能、编写难度（例如，产生无错误实现所需的小时数）、阅读难度（例如，代码行数及其微妙程度或非显而易见的正确性），以及你认为对读者（或你完成这项作业之前的自己）有趣的任何其他维度进行评估。如果适用，请包含任何测量结果。

   (c) 实现挑战。描述你发现代码中最令人头疼的部分，并解释原因。反思你是如何克服这些挑战的，以及是什么帮助你最终理解了那个困扰你的概念。你是如何尝试确保你的代码保持你的假设、不变量和前提条件的，以及你发现这容易还是困难？你是如何调试和测试你的代码的？

   (d) 剩余的bug。指出并尽可能解释代码中仍然存在的任何bug（或未处理的边缘情况）。

4. In your writeup, please also fill in the number of hours the assignment took you and any other comments.

5. Please let the course staff know ASAP of any problems at the lab session, or by posting a question on Ed. Good luck!

5 Extra Credit

Extra credit will be rewarded for improvements to the test suite. Add a test case to one of the files in the `tests` directory (e.g. `minnow/tests/recv_connect.cc`) that catches a **real bug** that somebody might reasonably make that isn't already caught by the existing test suite. Please submit your test as a Pull Request (it's okay to make this public) so we can take a look and decide whether to add it to the overall testsuite. (This opportunity will remain open—e.g. if you find a good additional test for the `Reassembler` in week 7, that's great too.)

4. 在你的报告里，请同时填写完成作业所花费的小时数以及任何其他评论。

5. 请尽快通知课程工作人员实验室期间遇到的问题，或在Ed上发布问题。祝你好运！

5 额外加分

若改进测试套件，将获得额外加分。向 `tests` 目录中的某个文件（例如 `minnow/tests/recv connect.cc`）添加一个测试用例，该用例能捕获一个真实且合理存在的、现有测试套件尚未覆盖的bug。请将您的测试用例作为Pull Request提交（公开提交也可以），以便我们查看并决定是否将其添加到整体测试套件中。（此机会将保持开放——例如，如果在第7周您为 `Reassembler` 找到一个良好的附加测试用例，那也很好。）