Lab Checkpoint 3: the TCP sender

Due: Sunday, Oct. 19, 11:59 p.m.

Collaboration Policy: Same as checkpoint 0. Please do not look at other students' 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`). In Checkpoints 1 and 2, you implemented the tools that translate *from* segments carried in unreliable datagrams *to* an incoming byte stream: the `Reassembler` and `TCPReceiver`.

Now, in Checkpoint 3, you'll implement the other side of the connection. The `TCPSender` is a tool that translates *from* an outbound byte stream *to* segments that will become the payloads of unreliable datagrams. This will complete your implementation of the Transmission Control Protocol (the implementations of which are arguably the world's most prevalent computer program, period). You'll use this to talk to a classmate and to peers across the Internet—real servers that speak TCP.

1 Getting started

Your implementation of a `TCPSender` will use the same Minnow library that you used in Checkpoints 0–2, 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 3 by running `git merge origin/check3-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/check3-startercode`.)
4. Make sure your build system is properly set up: `cmake -S . -B build`
5. Compile the source code: `cmake --build build`
6. Open and start editing the `writeups/check3.md` file. This is the template for your lab writeup and will be included in your submission.

实验检查点3：TCP发送方

截止日期：10月19日星期日，晚上11:59

合作政策：与检查点0相同。请勿查看其他学生的代码或这些作业过去版本的解决方案。请在实验报告中充分披露任何合作者或任何灰色地带——公开透明是最好的政策。

0 概述

建议：在实现前阅读整个实验文档。

在检查点0中，你实现了流控字节流的抽象(`ByteStream`)。在检查点1和2中，你实现了将不可靠数据报中的段转换成输入字节流的工具： `Reassembler` 和 `TCPReceiver`。

现在，在检查点3中，你将实现连接的另一端。 `TCPSender` 是一个将输出字节流转换成不可靠数据报有效载荷段的工具。这将完成你对传输控制协议（其实现可以说是世界上最普遍的计算机程序）的完整实现。你将使用它与同学以及互联网上的同行——那些说TCP的真实服务器——进行通信。

1 开始

你的 `TCPSender` 实现将使用你在检查点0-2中使用的相同Minnow库，并添加了额外的类和测试。开始使用：

1. 确保你已经将所有检查点1的解决方案提交了。请不要修改 `src` 目录顶层之外的任何文件，或者 `webget.cc`。否则，你可能难以合并检查点1的启动代码。
2. 在实验作业的代码库中，运行 `git fetch --all` 以获取实验作业的最新版本。
3. 通过运行 `git merge origin/check3-startercode` 下载 Checkpoint 3 的初始代码。（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/check3-startercode`。）
4. 确保你的构建系统已正确设置： `cmake -S . -B build`
5. 编译源代码： `cmake --build build`
6. 打开并开始编辑 `writeups/check3.md` 文件。这是你的实验报告模板，并将包含在你的提交中。

7. Reminder: please make frequent **small commits** in your local Git repository as you work. If you need help to make sure you're doing this right, please ask a classmate or the teaching staff for help. You can use the `git log` command to see your Git history.

2 Checkpoint 3: The TCP Sender

TCP is a protocol that reliably conveys a pair of flow-controlled byte streams (one in each direction) over unreliable datagrams. Two party participate in the TCP connection, and *each party* is a peer of the other. 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 "sender" part of TCP, responsible for reading from a `ByteStream` (created and written to by some sender-side application), and turning the stream into a sequence of outgoing TCP segments. On the remote side, a TCP receiver[1] transforms those segments (those that arrive—they might not all make it) back into the original byte stream, and sends acknowledgments and window advertisements back to the sender.

It will be your `TCPSender`'s responsibility to:

- Keep track of the receiver's window (receiving incoming `TCPReceiverMessage`s with their **ackno**s and **window size**s)
- Fill the window when possible, by reading from the `ByteStream`, creating new TCP segments (including SYN and FIN flags if needed), and sending them. The sender should *keep sending segments* until either the window is full or the outbound `ByteStream` has nothing more to send.
- Keep track of which segments have been sent but not yet acknowledged by the receiver—we call these "outstanding" segments
- Re-send outstanding segments if enough time passes since they were sent, and they haven't been acknowledged yet

⋆*Why am I doing this?* The basic principle is to send whatever the receiver will allow us to send (filling the window), and keep retransmitting until the receiver acknowledges each segment. This is called "automatic repeat request" (ARQ). The sender divides the byte stream up into segments and sends them, as much as the receiver's window allows. Thanks to your work last week, we know that the remote TCP receiver can reconstruct the byte stream as long as it receives each index-tagged byte at least once—no matter the order. The sender's job is to make sure the receiver gets each byte at least once.

[1] It's important to remember that the receiver can be *any* implementation of a valid TCP receiver—it won't necessarily be your own `TCPReceiver`. One of the valuable things about Internet standards is how they establish a common language between endpoints that may otherwise act very differently.

7. 提醒：请在你本地的Git仓库中频繁提交小改动。如果你需要帮助确保操作正确，请向同学或助教寻求帮助。你可以使用 `git log` 命令查看你的Git历史记录。

2 检查点3：TCP发送方

TCP 是一种协议，它通过不可靠的数据报可靠地传输一对流量控制的字节流（每个方向一个）。TCP 连接中有两方参与，每一方都是对方的对等方。每一方同时充当"发送者"（发送自己的传出字节流）和"接收者"（接收传入的字节流）。

本周，你将实现TCP的"发送方"部分，负责从`ByteStream`（由某些发送端应用程序创建和写入）读取数据，并将数据流转换为一系列外发的TCP段。在远程端，TCP接收方[1]将这些段（那些到达的——它们可能并非全部到达）转换回原始字节流，并向发送方发送确认号和窗口广告。

你的 `TCPSender`的责任是：

- 持续跟踪接收方的窗口（接收带有确认号和窗口大小的入站 `TCPReceiverMessage`）
- 在可能的情况下填充窗口，通过从 `ByteStream`读取、创建新的TCP段（如有需要包含SYN和FIN标志），并发送它们。发送方应持续发送段，直到窗口已满或出站 `ByteStream` 无更多内容发送。
- 持续跟踪已发送但尚未被接收方确认的段——我们称这些为"未确认"段
- 如果自发送以来经过足够时间且尚未被确认，则重新发送未确认段

⋆我为什么要做这个？基本原理是发送接收器允许我们发送的内容（填满窗口），并持续重传，直到接收器确认每个分段。这被称为"自动重传请求"（ARQ）。发送方将字节流分割成多个分段并发送，尽可能多地发送接收器窗口允许的内容。得益于你上周的工作，我们知道远程TCP接收器只要至少接收到每个索引标记的字节一次——无论顺序——就能重建字节流。发送方的任务是确保接收器至少接收到每个字节一次。

[1] 重要的是要记住，接收方可以是任何有效的TCP接收方实现——它不一定是你自己的TCPReceiver。互联网标准的价值之一在于它们如何为可能行为差异很大的端点之间建立通用语言。

2.1 How does the TCPSender know if a segment was lost?

Your TCPSender will be sending a bunch of TCPSenderMessages. Each will contain a (possibly-empty) substring from the outgoing ByteStream, indexed with a sequence number to indicate its position in the stream, and marked with the SYN flag at the beginning of the stream, and FIN flag at the end.

In addition to *sending* those segments, the TCPSender also has to *keep track of* its outstanding segments until the sequence numbers they occupy have been fully acknowledged. Periodically, the owner of the TCPSender will call the TCPSender's tick method, indicating the passage of time. The TCPSender is responsible for looking through its collection of outstanding TCPSenderMessages and deciding if the oldest-sent segment has been outstanding for too long without acknowledgment (that is, without *all* of its sequence numbers being acknowledged). If so, it needs to be retransmitted (sent again).

Here are the rules for what "outstanding for too long" means.[2] You're going to be implementing this logic, and it's a little detailed, but we don't want you to be worrying about hidden test cases trying to trip you up or treating this like a word problem on the SAT. We'll give you some reasonable unit tests this week, and fuller integration tests in Lab 4 once you've finished the whole TCP implementation. As long as you pass those tests 100% and your implementation is reasonable, you'll be fine.

⋆*Why am I doing this?* The overall goal is to let the sender detect when segments go missing and need to be resent, in a timely manner. The amount of time to wait before resending is important: you don't want the sender to wait too long to resend a segment (because that delays the bytes flowing to the receiving application), but you also don't want it to resend a segment that was going to be acknowledged if the sender had just waited a little longer—that wastes the Internet's precious capacity.

1. Every few milliseconds, your TCPSender's tick method will be called with an argument that tells it how many milliseconds have elapsed since the last time the method was called. Use this to maintain a notion of the total number of milliseconds the TCPSender has been alive. **Please don't try to call any "time" or "clock" functions** from the operating system or CPU—the tick method is your *only* access to the passage of time. That keeps things deterministic and testable.

2. When the TCPSender is constructed, it's given an argument that tells it the "initial value" of the **retransmission timeout** (RTO). The RTO is the number of milliseconds to wait before resending an outstanding TCP segment. The value of the RTO will change over time, but the "initial value" stays the same. The starter code saves the "initial value" of the RTO in a member variable called initial_RTO_ms_.

[2]These are based on a simplified version of the "real" rules for TCP: RFC 6298, recommendations 5.1 through 5.6. The version here is a bit simplified, but your TCP implementation will still be able to talk with real servers on the Internet.

2.1 TCPSender如何知道一个分段是否丢失?

你的 TCPSender 将发送一批 TCPSenderMessage。每个 TCPSenderMessage将包含一个（可能为空的）从传出 ByteStream的子字符串，使用序列号索引以指示其在流中的位置，并在流的开始处标记SYN标志，在流的结束处标记FIN标志。

除了发送这些分段， TCPSender 还必须跟踪其未确认的分段，直到它们占用的序列号全部被确认。定期地， TCPSender 的所有者会调用 TCPSender的 tick 方法，表示时间的流逝。 TCPSender 负责检查其未确认TCPSenderMessage的集合，并决定最早发送的分段是否已过长时间未获确认（即，其所有序列号未被全部确认）。如果是，则需要重新传输（再次发送）。

以下是“过长时间未确认”的定义规则。[2] 你将要实现这个逻辑，它有点复杂，但我们不希望你担心隐藏的测试用例试图让你出错，或者把它当作SAT考试中的文字题。本周我们会给你一些合理的单元测试，在完成整个TCP实现后，在实验4中会提供更完整的集成测试。只要你100%通过那些测试，并且你的实现是合理的，你就能顺利通过。

⋆我为什么要做这个？总体目标是让发送者检测到段
丢失并需要重新发送，应及时。等待的时间
重发前很重要：你不想发送者等待太长时间来重发一个片段（因为这会延迟字节流向接收应用程序），但你也不想重发一个片段，如果发送者只是再等待一小会儿就会被确认——那会浪费互联网宝贵的容量。

1. 每隔几毫秒，你的 TCPSender 的 tick 方法就会被调用，传入一个参数，告诉它自上次调用该方法以来已经过去了多少毫秒。利用这一点来维护一个 TCPSender 存活的总毫秒数的概念。请不要尝试调用操作系统或 CPU 中的任何“时间”或“时钟”函数—— tick 方法是你访问时间流逝的唯一途径。这能保持事情的确定性，并使其可测试。

2. 当 TCPSender 被构建时，它会被赋予一个参数，告知其重传超时 (RTO) 的“初始值”。RTO 是在重新发送一个未完成的 TCP 段之前需要等待的毫秒数。RTO 的值将随时间变化，但“初始值”保持不变。启动代码将RTO的“初始值”保存在名为 initial RTO ms 的成员变量中。

[2] 这些基于 TCP 的“真实”规则的简化版本：RFC 6298，建议 5.1 至 5.6。这里的版本稍显简化，但您的 TCP 实现仍然能够与互联网上的真实服务器进行通信。

3. You'll implement the retransmission **timer**: an alarm that can be started at a certain time, and the alarm goes off (or "expires") once the RTO has elapsed. We emphasize that this notion of time passing comes from the `tick` method being called—not by getting the actual time of day.

4. Every time a segment containing data (nonzero length in sequence space) is sent (whether it's the first time or a retransmission), if the timer is not running, **start it running** so that it will expire after RTO milliseconds (for the current value of RTO). By "expire," we mean that the time will run out a certain number of milliseconds in the future.

5. When all outstanding data has been acknowledged, **stop** the retransmission timer.

6. If `tick` is called and the retransmission timer has expired:

   (a) Retransmit the *earliest* (lowest sequence number) segment that hasn't been fully acknowledged by the TCP receiver. You'll need to be storing the outstanding segments in some internal data structure that makes it possible to do this.

   (b) **If the window size is nonzero:**

      i. Keep track of the number of *consecutive* retransmissions, and increment it because you just retransmitted something. Your `TCPConnection` will use this information to decide if the connection is hopeless (too many consecutive retransmissions in a row) and needs to be aborted.

      ii. Double the value of RTO. This is called "exponential backoff"—it slows down retransmissions on lousy networks to avoid further gumming up the works.

   (c) Reset the retransmission timer and start it such that it expires after RTO milliseconds (taking into account that you may have just doubled the value of RTO!).

7. When the receiver gives the sender an `ackno` that acknowledges the successful receipt of *new* data (the `ackno` reflects an absolute sequence number bigger than any previous `ackno`):

   (a) Set the RTO back to its "initial value."

   (b) If the sender has any outstanding data, restart the retransmission timer so that it will expire after RTO milliseconds (for the current value of RTO).

   (c) Reset the count of "consecutive retransmissions" back to zero.

You might choose to implement the functionality of the retransmission timer in a separate class, but it's up to you. If you do, please add it to the existing files (`tcp_sender.hh` and `tcp_sender.cc`).

3. 你将实现重传计时器：一个可以在特定时间启动的闹钟，当重传超时（RTO）经过后闹钟就会响起（或“过期”）。我们强调，这种时间流逝的概念来自于 `tick` 方法的调用——而不是通过获取实际的时间。

4. 每次发送包含数据（序列空间中非零长度）的片段（无论是第一次发送还是重传），如果计时器没有运行，就启动它运行，以便它在RTO毫秒后（对于当前的RTO值）过期。通过“过期”我们指的是，时间将在未来某数毫秒后用完。

5. 当所有未确认的数据都已确认时，停止重传计时器。

6. 如果 `tick` 被调用且重传计时器已过期:

   (a) 重传TCP接收器尚未完全确认的最早（序列号最低）片段。你需要将未确认的片段存储在某种内部数据结构中，以便能够做到这一点。

   (b) 如果窗口大小不为零:

      i. 保持跟踪连续重传的次数，并且增加它，因为你刚刚重传了某些内容。你的 `TCPConnection` 将使用这些信息来决定连接是否绝望（连续重传次数过多）并且需要中止。

      ii. 将 RTO 的值加倍。这被称为“指数退避”——它通过减慢在糟糕网络上的重传速度来避免进一步导致系统卡顿。

   (c) 重置重传计时器并启动它，使其在 RTO 毫秒后过期
   重传超时 (考虑到你可能刚刚将 RTO 的值翻了一倍！)

7. 当接收者向发送者发送一个 `ackno` 确认号，表示已成功收到新数据（该 `ackno` 反映了一个绝对序列号，其值大于任何之前的 `ackno`）：

   (a) 将 RTO 重置为其“初始值”。

   (b) 如果发送方有任何未处理的数据，重新启动重传计时器，使其在 RTO 毫秒后过期（使用当前 RTO 的值）。

   (c) 将“连续重传”计数重置为零。

你可以选择将重传计时器的功能实现在一个单独的类中，但这取决于你。如果你这样做，请将其添加到现有的文件中（`tcp sender.hh` 和`tcp sender.cc`）。

2.2 Implementing the TCP sender

Okay! We've discussed the basic idea of *what* the TCP sender does (given an outgoing `ByteStream`, split it up into segments, send them to the receiver, and if they don't get acknowledged soon enough, keep resending them). And we've discussed *when* to conclude that an outstanding segment was lost and needs to be resend.

Now it's time for the concrete interface that your `TCPSender` will provide. There are four important events that it needs to handle:

1. `void push( const TransmitFunction& transmit );`

   The `TCPSender` is asked to *fill the window* from the outbound byte stream: it reads from the stream and sends as many `TCPSenderMessage`s as possible, *as long as there are new bytes to be read and space available in the window*. It sends them by calling the provided `transmit()` function on them.

   You'll want to make sure that every `TCPSenderMessage` you send fits fully inside the receiver's window. Make each *individual* message as big as possible, but no bigger than the value given by `TCPConfig::MAX_PAYLOAD_SIZE`.

   You can use the `TCPSenderMessage::sequence_length()` method to count the total number of sequence numbers occupied by a segment. Remember that the SYN and FIN flags also occupy a sequence number each, which means that *they occupy space in the window*.

   ⋆*What should I do if the window size is zero?* If the receiver has announced a window size of zero, the `push` method should pretend like the window size is **one**. The sender might end up sending a single byte that gets rejected (and not acknowledged) by the receiver, but this can also provoke the receiver into sending a new acknowledgment segment where it reveals that more space has opened up in its window. Without this, the sender would never learn that it was allowed to start sending again.
   This is the only special-case behavior your implementation should have for the case of a zero-size window. The `TCPSender` shouldn't actually *remember* a false window size of 1. The special case is only within the push method. Also, N.B. that even if the window size is one (or 20, or 200), the window might still be **full**. A "full" window is not the same as a "zero-size" window.

2. `void receive( const TCPReceiverMessage& msg );`

   A message is received from the receiver, conveying the new left (= `ackno`) and right (= `ackno` + `window_size`) edges of the window. The `TCPSender` should look through its collection of outstanding segments and remove any that have now been fully acknowledged (the `ackno` is greater than all of the sequence numbers in the segment).

3. `void tick( uint64_t ms_since_last_tick, const TransmitFunction& transmit );`

2.2 实现TCPSender

好的！我们已经讨论了TCP发送器的基本工作原理（给定一个外发的`ByteStream`，将其拆分成多个段，发送给接收器，如果它们没有在足够短的时间内得到确认，就继续重发它们）。我们也讨论了何时确定一个未完成的段丢失了，需要重新发送。

现在，是时候提供你的助教将实现的具体接口了。它需要处理四个重要的事件：

1. `void push( const TransmitFunction& transmit );`

   助教被要求从出站字节流中填充窗口：它从流中读取，并发送尽可能多的 `TCPSenderMessage`，只要还有新的字节可读且窗口中有可用空间。它通过在这些 `TCPSenderMessage`上调用提供的 `transmit()` 函数来发送它们。

   你需要确保发送的每个 `TCPSenderMessage` 完全适配接收方的窗口。让每条消息尽可能大，但不能超过 `TCPConfig::MAX PAYLOAD SIZE` 给定的值。

   你可以使用 `TCPSenderMessage::sequence length()` 方法来统计一个段占用的序列号总数。请记住，SYN 和 FIN 标志各占用一个序列号，这意味着它们会占用窗口空间。

   ⋆ 如果窗口大小为零该怎么办？如果接收方宣布窗口大小为零， `push` 方法应假装窗口大小为 1。发送方可能会发送一个字节，但接收方会拒绝（且不确认）该字节，但这也可能促使接收方发送一个新的确认段，其中会透露其窗口中已出现更多可用空间。否则，发送方将永远不会知道可以重新开始发送。

   这是你的实现针对零大小窗口的唯一特殊情况行为。 `TCPSender` 实际上不应记住一个虚假的 1 大小窗口。特殊情况仅在推送方法中有效。此外，请注意，即使窗口大小为 1（或 20，或 200），窗口仍可能已满。

2. `void receive( const TCPReceiverMessage& msg );`

   收到来自接收者的消息，传达了窗口的新左边缘（= `ackno`）和右边缘（= `ackno` + `window size`）。 `TCPSender` 应检查其未完成段集合，并移除所有已被完全确认的段（即 `ackno` 大于段中所有序列号）。

3. `void tick( uint64 t ms since last tick, const TransmitFunction& transmit );`

Time has passed — a certain number of milliseconds since the last time this method was called. The sender may need to retransmit an outstanding segment; it can call the **`transmit()`** function to do this. (Reminder: please don't try to use real-world "clock" or "gettimeofday" functions in your code; the only reference to time passing comes from the `ms_since_last_tick` argument.)

4. `TCPSenderMessage make_empty_message() const;`

   The `TCPSender` should generate and send a zero-length message with the sequence number set correctly. This is useful if the peer wants to send a `TCPReceiverMessage` (e.g. because it needs to acknowledge something from the peer's sender) and needs to generate a `TCPSenderMessage` to go with it.

   Note: a segment like this one, which occupies no sequence numbers, doesn't need to be kept track of as "outstanding" and won't ever be retransmitted.

To complete Checkpoint 3, please review the full interface in **`src/tcp_sender.hh`** implement the complete `TCPSender` public interface in the `tcp_sender.hh` and `tcp_sender.cc` files. We expect you'll want to add private methods and member variables, and possibly a helper class.

2.3 FAQs and special cases

- *What should my* `TCPSender` *assume as the receiver's window size before the receive method informs it otherwise?*

  One.

- *What do I do if an acknowledgment only* partially *acknowledges some outstanding segment? Should I try to clip off the bytes that got acknowledged?*

  A TCP sender *could* do this, but for purposes of this class, there's no need to get fancy. Treat each segment as fully outstanding until it's been fully acknowledged—all of the sequence numbers it occupies are less than the `ackno`.

- *If I send three individual segments containing "a," "b," and "c," and they never get acknowledged, can I later retransmit them in one big segment that contains "abc"? Or do I have to retransmit each segment individually?*

  Again: a TCP sender *could* do this, but for purposes of this class, no need to get fancy. Just keep track of each outstanding segment individually, and when the retransmission timer expires, send the earliest outstanding segment again.

- *Should I store empty segments in my "outstanding" data structure and retransmit them when necessary?*

  No—the only segments that should be tracked as outstanding, and possibly retransmitted, are those that convey some data—i.e. that consume some length in sequence space. A segment that occupies no sequence numbers (no SYN, payload, or FIN) doesn't need to be remembered or retransmitted.

时间已经过去——自上次调用此方法以来经过了一定的毫秒数。发送者可能需要重新传输一个悬而未决的片段；它可以调用`transmit()`函数来完成这个操作。（提醒：请不要在代码中尝试使用现实世界的“时钟”或“gettimeofday”函数；唯一的时间流逝参考仅来自 `ms since last tick` 参数。）

4. `TCPSenderMessage make empty message() const;`

   助教应生成并发送一个长度为零的消息，并将序列号设置正确。如果对端想发送一个 `TCPReceiverMessage`（例如，因为它需要确认来自对端发送者的一些内容），并且需要生成一个 `TCPSenderMessage` 与之配合，那么这是有用的。

   注意：像这样不占用序列号的段，不需要作为“未完成”进行跟踪，也永远不会被重传。

要完成第3个检查点，请复习 `src/tcp sender.hh` 中的完整接口，并在 `tcp_sender.hh` 和 `tcp sender.cc` 文件中实现完整的 `TCPSender` 公共接口。我们预计您可能需要添加私有方法和成员变量，并且可能需要一个辅助类。

2.3 常见问题解答和特殊情况

- 在接收方法通知我之前，我应该将 `TCPSender` 假设为接收者的窗口大小吗？方法会告知我其他情况？

  One.

- 如果确认号只部分确认了某些未处理的数据段，我该怎么办？我应该尝试剪掉已确认的字节吗？

  TCP 发送者可以这样做，但就本课程而言，无需过于复杂。将每个数据段视为完全未处理，直到它被完全确认——它占用的所有序列号都小于 `ackno`。

- 如果我发送三个包含*“a”*、*“b”*和*“c”*的独立数据段，并且它们从未已确认，我稍后能否将它们重传为一个包含*“abc”*的大段？或者必须单独重传每个段？

  再说一次：TCP发送者可以这样做，但就本课程而言，无需花哨。只需单独跟踪每个未确认的段，当重传计时器到期时，再次发送最早的未确认段。

- 我是否应该在“未确认”数据结构中存储空段并在需要时重传？

  不——只有那些传输数据（即占用序列空间长度）的片段才应该被跟踪为未完成状态，并可能需要重传。一个不占用任何序列号（没有 SYN、有效载荷或 FIN）的片段不需要被记住或重传。

- *Where can I read if there are more FAQs after this PDF comes out?*
  Please check the website (https://cs144.github.io/lab_faq.html) and Ed regularly.

3 Development and debugging advice

1. Implement the `TCPSender`'s public interface (and any private methods or functions you'd like) in the file `tcp_sender.cc`. You may add any private members you like to the `TCPSender` class in `tcp_sender.hh`.
2. You can test your code with `cmake --build build --target check3`.
3. Please re-read the section on "using Git" in the Checkpoint 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 it hard to follow your code.

4 Hands-on activity

Congratulations—you have made a fully working implementation of the Transmission Control Protocol, implementations of which are arguably the most prevalent computer program on the planet. It's time to take a victory lap! You'll communicate with Linux's TCP and with a lab partner, and then you'll modify your webget (from checkpoint 0) to use *your* TCP implementation. In your writeup, describe what you did, answer the questions below, and try to find something interesting to discuss!

4.1 Experiments within your own VM

We've given you a client program (`./build/apps/tcp_ipv4`) that uses your TCPSender and TCPReceiver to speak TCP-over-IP over the Internet.[3] We've also given you a similar program (`./build/apps/tcp_native`) that uses a Linux `TCPSocket`.

The big question: Can your TCP implementation (`tcp_ipv4`) interoperate with Linux's TCP (`tcp_native`)?

[3]If you're curious how this program works, the `tcp_peer.hh` and `tcp_over_ip.cc` files are probably the interesting part of how we glued your TCPSender/TCPReceiver into a conforming TCP peer.

•如果这份*PDF*发布后还有更多常见问题解答，我可以在哪里阅读？
请检查网站（https://cs144.github.io/lab faq.html）并定期查看 Ed 的内容。

3 开发与调试建议

1. 在文件 `tcp sender.cc`中实现 `TCPSender`的公共接口（以及你希望添加的任何私有方法或函数）。你可以将任何你喜欢的私有成员添加到 `tcp sender.hh`中的 `TCPSender` 类。

2. 你可以用 `cmake --build build --target check3`测试你的代码。

3. 请重新阅读检查点0文档中关于“使用Git”的部分，并记住要保留代码在它被分发到的Git仓库的 `main` 分支上。提交小的更改，使用良好的提交信息，以标识所做的更改及其原因。

4. 请努力让你的代码对助教评阅风格时易于阅读。为变量使用合理且清晰的命名约定。使用注释来解释复杂或微妙的代码部分。使用“防御式编程”——明确检查函数的前置条件或不变式，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代码难以理解。

4 实践活动

恭喜你——你已经完成了一个完全可工作的传输控制协议（TCP）的实现，其实现可以说是地球上最普遍的计算机程序之一。是时候庆祝一下了！你将与Linux的TCP和一个实验伙伴进行通信，然后你将修改从检查点0获得的webget，使其使用你的TCP实现。在你的实验报告中，描述你做了什么，回答下面的问题，并尝试找到一些有趣的东西来讨论！

4.1 在你自己的虚拟机中进行实验

我们已经为你提供了一个客户端程序（`./build/apps/tcp ipv4`），它使用你的 TCP 发送器（TCPSender）和 TCP 接收器（TCPReceiver）在互联网上通过 TCP-over-IP 进行通信。[3] 我们还为你提供了一个类似的程序（`./build/apps/tcp native`），它使用 Linux `TCPSocket`。

关键问题：你的 TCP 实现 (`tcp ipv4`) 能否与 Linux 的 TCP (`tcp native`) 互操作?

[3]如果你想知道这个程序是如何工作的，那么 tcp peer.hh 和 tcp over ip.cc 文件可能是我们如何将你的 TCPSender/TCPReceiver 集成到一个符合规范的 TCP peer 中的有趣部分。

4.1.1 Have Linux's TCP talk to itself

- First, let's do the boring part of making sure Linux's TCP implementation can talk to itself. Run Linux's TCP as a "server" (the peer that waits for an incoming SYN segment), listening on port 9090. On your VM, run: `./build/apps/tcp_native -l 0 9090`
- Next, try using Linux's TCP as the "client": the peer that initiates the connection by sending the first SYN segment to the server. In another terminal window on your VM, run: `./build/apps/tcp_native 169.254.144.1 9090`
- If all goes well, the "server" will print something like `DEBUG: New connection from 169.254.144.1:36568` and the "client" will print something like `DEBUG: Connecting to 169.254.144.1:9090... DEBUG: Successfully connected to 169.254.144.1:9090.`
- Try typing into each window, and you will see the same bytes on the other window.
- To end a stream, type ctrl-D (on a line by itself) to close the ByteStream Writer in *that* direction. If all goes well, you'll see `Outbound stream...finished` on the terminal where you typed the ctrl-D, and `Inbound stream...finished` on the other terminal. Notice that the other peer can keep sending to the "closed" peer—each direction of the stream can be closed independently, without preventing the other direction from continuing.
- Now end the stream in the second direction by typing ctrl-D (on a line by itself) in the other terminal. If all went well, both programs will quit and bring you back to the command line in both terminals. This indicates the TCP connection has finished in both directions (as discussed in class, Linux will "linger" in the background before reusing one of the port numbers to reduce the chance of a "two general's problem").

4.1.2 Have *your* TCP talk to Linux's

Repeat the above steps, but connect *your* TCP implementation to Linux's. **First,** run `sudo ./scripts/tun.sh start 144` to give your implementation permission to send raw Internet datagrams without needing to be root. You'll have to rerun this command any time you reboot your VM.

Then, rerun the above experiment, replacing *one* of the programs (the client or server) with `tcp_ipv4` (which is *your* TCP implementation). Does the connection still get established as before, and can each peer still type at the other and have the text appear on the other peer's window? If so, pat yourself on the back (and we'll shake your hand)—you've earned it! If not. . . time to start debugging. You can capture the TCP segments with a command like `sudo rm -f /tmp/capture.raw; sudo tcpdump -n -w /tmp/capture.raw -i tun144 --print --packet-buffered`; the resulting `/tmp/capture.raw` file can be visualized in wireshark as before.

After you've typed a little in each direction, try closing one of the ByteStreams and keep typing a little in the other direction. Do both programs quit cleanly after both streams have

4.1.1 让Linux的TCP与自己通信

- 首先，让我们做些枯燥的工作，确保Linux的TCP实现能够与自己通信。将Linux的TCP作为"服务器"（等待传入SYN段的端点），监听9090端口。在你的虚拟机中运行：`./build/apps/tcp native -l 0 9090`
- 接下来，尝试将Linux的TCP作为"客户端"：通过向服务器发送第一个SYN段来发起连接。在你的虚拟机的另一个终端窗口中运行：`./build/apps/tcp native 169.254.144.1 9090`
- 如果一切顺利，"服务器"将打印类似 `DEBUG: New connection from169.254.144.1:36568` 的内容，而"客户端"将打印类似 `DEBUG: Connectingto 169.254.144.1:9090... DEBUG: Successfull`的内容 `y connected to 169.254.144.1:9090.`
- 试着在每个窗口中输入，你会在另一个窗口看到相同的字节。
- 要结束一个流，在单独的一行中输入 ctrl-D 来关闭该方向的 ByteStream 写入器。如果一切顺利，你会在你输入 ctrl-D 的终端看到 `Outbound stream...finished`，在另一个终端看到 `Inbound stream...finished`。请注意，另一个对等方可以继续向"已关闭"的对等方发送——流的方向可以独立关闭，而不会阻止另一个方向继续进行。
- 现在在另一个终端中通过在单独的一行中输入 ctrl-D 来结束第二个方向的流。如果一切顺利，两个程序都会退出，并将你返回到两个终端的命令行。这表明 TCP 连接在两个方向上都已完成（正如课堂上讨论的，Linux 会"挂起"在后台，然后重用其中一个端口号，以减少"两个将军问题"的机会）。

4.1.2 让你的 TCP 与 Linux 的

重复上述步骤，但将你的 TCP 实现连接到 Linux 的。首先，运行 `sudo ./scripts/tun.sh start 144` 来给你的实现权限，以便无需成为 root 就发送原始互联网数据报。你将不得不每次重启你的 VM 时都重新运行此命令。

然后，重新运行上述实验，将其中一个程序（客户端或服务器）替换为`tcp ipv4`（即你的TCP实现）。连接是否仍然像之前一样建立，并且每个对等方是否仍然可以输入文本并在另一个对等方的窗口中显示？如果是，请给自己鼓鼓掌（我们将与你握手）——你应得的！如果不是……是时候开始调试了。你可以使用类似 sudo rm -f /tmp/capture.raw; sudo tcpdump -n -w /tmp/capture.raw -i tun144 --print --packet-buffered ; 生成的 `/tmp/capture.raw` 文件可以像之前一样在wireshark中可视化。

在各个方向都输入一些内容后，尝试关闭其中一个字节流，并在另一个方向继续输入一点。当两个流都完成后，两个程序是否都能干净地退出？

finished with a ctrl-D? They should—although you may need to see `tcp_ipv4` wait a little to reduce the chance of a "two general's problem." When does it need to wait (when it's the first to close or the second to close)? Does this match what was discussed in class?

4.1.3 Try to pass the "one megabyte challenge"

Once it looks like you can have a basic conversation, try sending a file between `tcp_ipv4` (your TCP) and `tcp_native` (Linux's TCP).

To **create** a random file that's 12345 bytes as "/tmp/big.txt":

```
dd if=/dev/urandom bs=12345 count=1 of=/tmp/big.txt
```

You can choose the direction of transmission—i.e. whether the client or server is the one to send the file.

To have the **server** send the file as soon as it accepts an incoming connection, redirect standard input to read from the file:

```
./build/apps/tcp_native -l 0 9090 < /tmp/big.txt
```

To have the client receive the file, close off its outgoing stream by redirecting from `/dev/null`, and redirect standard output to a second file named "/tmp/big-received.txt":

```
</dev/null ./build/apps/tcp_ipv4 169.254.144.1 9090 > /tmp/big-received.txt
```

Or to have the **server** receive the file:

```
</dev/null ./build/apps/tcp_native -l 0 9090 > /tmp/big-received.txt
```

Or to have the **client** send the file:

```
./build/apps/tcp_ipv4 169.254.144.1 9090 < /tmp/big.txt
```

To compare two files and make sure they're the same:
`sha256sum /tmp/big.txt` or `sha256sum /tmp/big-received.txt`

If the SHA-256 hashes match, you can be almost certain the file was transmitted correctly.

Try this with a tiny file (12 bytes), then 65534 bytes (a little less than 2^{16}), then 65537 bytes (a little omre than 2^{16}), then 200000 bytes, then the full megabyte (1000000 bytes). If they all match, give yourself an even bigger pat on the back! If not... time to debug (possibly with tcpdump and wireshark as described above).

4.2 Reach out and talk to a friend

If everything works above, try communicating with a labmate over the Internet! One of you will run `tcp_native` as a server, as above. The other will run `tcp_ipv4` as the client, connecting to the labmate's address on the CS144 private network (10.144....).

使用 ctrl-D 完成操作？它们应该能——尽管你可能需要等待一会儿以降低“两个将军问题”发生的概率。它需要等待什么时候（是第一个关闭还是第二个关闭）？这与课堂上讨论的内容是否一致?

4.1.3 尝试通过“一兆字节挑战”

一旦看起来可以进行基本对话，尝试在 `tcp ipv4`(你的TCP) 和 `tcp native` (Linux的TCP) 之间发送文件。

To **create** a random file that's 12345 bytes as "/tmp/big.txt":

```
dd if=/dev/urandom bs=12345 count=1 of=/tmp/big.txt
```

你可以选择传输方向——即客户端或服务器是发送文件的一方。

要让服务器在接受传入连接后立即发送文件，请将标准输入重定向为从文件读取:

```
./build/apps/tcp_native -l 0 9090 < /tmp/big.txt
```

要让客户端接收文件，请通过重定向来自 `/dev/null` 来关闭其出站流，并将标准输出重定向到名为“/tmp/big-received.txt”的第二个文件:

```
</dev/null ./build/apps/tcp_ipv4 169.254.144.1 9090 > /tmp/big-received.txt
```

或者，要让服务器接收文件:

```
</dev/null ./build/apps/tcp_native -l 0 9090 > /tmp/big-received.txt
```

或者，要让客户端发送文件:

```
./build/apps/tcp_ipv4 169.254.144.1 9090 < /tmp/big.txt
```

要比较两个文件并确保它们相同:
`sha256sum /tmp/big.txt` or `sha256sum /tmp/big-received.txt`

如果 SHA-256 哈希值匹配，你可以几乎确定文件已正确传输。

尝试用一个小文件（12 字节），然后是 65534 字节（略小于 2^{16}），然后是 65537 字节（略大于 2^{16}），然后是 200000 字节，最后是完整的兆字节（1000000 字节）。如果它们都匹配，给自己一个更大的表扬！如果不行……是时候调试了（可能需要像上面描述的那样使用 tcpdump 和 wireshark）。

4.2 主动联系朋友

如果以上所有内容都能正常工作，请尝试通过互联网与助教进行通信！你们中的一人将运行 `tcp native` 作为服务器，如上所述。另一人将运行 `tcp ipv4` 作为客户端，连接到 CS144 私有网络（10.144....）上的助教的地址。

Can you type to each other and successfully end the two streams cleanly? And if so, can you pass the one-megabyte challenge (sending a random 1000000-byte file successfully over the Internet to your labmate's VM, with the SHA-256 hashes matching perfectly on both sides)? If so, congratulations. . . now trade places and try sending the file in the other direction!

What's the biggest file that you have the patience to successfully send to your labmate? In your lab report, include the sizes of the two files (the output of `ls -l /tmp/big.txt` for the sender and `ls -l /tmp/big-received.txt` for the receiver) and the results of `sha256sum /tmp/big.txt` (on the sender's VM) and `sha256sum /tmp/big-received.txt` (on the receiver's).

4.3 webget revisited

Remember your `webget.cc` that you wrote in Checkpoint 0? It used a TCP implementation (`TCPSocket`) provided by the Linux kernel. We'd like you to switch it to use your own TCP implementation without changing anything else. We think that all you'll need to do is:

- Replace `#include "socket.hh"` with `#include "tcp_minnow_socket.hh"`.
- Replace the `TCPSocket` type with `CS144TCPSocket`.
- At the end of your `get_URL()` function, add a call to `socket.wait_until_closed()`.

> ⋆*Why am I doing this?* Normally the Linux kernel takes care of waiting for TCP connections to reach "clean shutdown" (and give up their port reservations) even after user processes have exited. But because your TCP implementation is all in user space, there's nothing else to keep track of the connection state except your program. Adding this call makes the socket wait until the connection is fully closed.

Recompile, and run `make check_webget` to confirm that you've gone full-circle: you've written a basic Web fetcher on top of *your own complete TCP "stack"*, and it still successfully talks to a real webserver. If you have trouble, try running the program manually: `./build/apps/webget cs144.keithw.org /hasher/xyzzy`. You'll get some debugging output on the terminal that may be helpful.

5 Submit

1. In your submission, please only make changes to the `.hh` and `.cc` files in the `src` directory (and `apps/webget.cc`). 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.

你们能互相输入文字并成功清理两个流吗？如果可以，你们能通过互联网成功发送一个随机 1000000 字节的大文件给助教的虚拟机，并且两边的 SHA-256 哈希值完全匹配吗？如果可以，恭喜。现在交换位置，尝试从另一个方向发送文件！

你最有耐心成功发送给实验伙伴的最大文件是什么？在你的实验报告中，请包含两个文件的大小（发送方的 `ls -l /tmp/big.txt`输出和接收方的 `ls -l /tmp/big-received.txt` 输出），以及发送方虚拟机上的`sha256sum /tmp/big.txt` 结果和接收方上的 `sha256sum /tmp/big-received.txt`结果。

4.3 webget再探

还记得你在检查点0中写的 `webget.cc` 吗？它使用了Linux内核提供的TCP实现（`TCPSocket`）。我们希望你能将其切换为使用你自己的TCP实现，而不改变其他任何东西。我们认为你只需要做以下几件事：

- 用 `#include "tcp minnow socket.hh"`替换 `#include "socket.hh"`.
- 将 `TCPSocket` 类型替换为 `CS144TCPSocket`。
- 在你的 `get URL()` 函数末尾，添加对 `socket.wait until closed()`的调用。.

> ⋆我为什么要这样做？通常情况下，Linux 内核会负责等待 TCP 连接达到“干净关闭”（并且放弃端口预留），即使用户进程已经退出。但由于你的 TCP 实现完全在用户空间中，除了你的程序之外，没有其他东西可以跟踪连接状态。添加这个调用会使套接字等待连接完全关闭。

重新编译，并运行 `make check webget` 来确认你是否回到了起点：你已经在自己的完整 TCP “栈”之上编写了一个基本的 Web 获取器，并且它仍然成功地与一个真实的 Web 服务器通信。如果你遇到问题，可以尝试手动运行程序：./build/apps/webget cs144.keithw.org / hasher/xyzzy 。你会在终端上看到一些调试输出，这可能有助于解决问题。

5 提交

1. 在你的提交中，请仅修改 `src`目录（以及 `apps/webget.cc`）中的 `.hh` 和 `.cc` 文件。在这些文件中，请根据需要自由添加私有成员，但请不要更改任何类的公共接口。

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 check3` (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/check3.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.

   (e) **Hands-on Activity.** Include answers to the questions and some thoughtful commentary on the hands-on activity above.

4. 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 a lab session, or by posting a question on Ed. Good luck!

2. 提交任何作业前，请按顺序运行以下命令：

   (a) 确保您已将所有更改提交到Git仓库。您可以使用 `git status` 来确保没有未完成的更改。记住：边编写代码边提交小批量更改。

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

   (c) `cmake --build build --target check3` (以确保自动测试通过)

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

3. 使用 `writeups/check3.md`编写报告。该文件应是一个大约20到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。描述代码中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中继承的内容。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帮助其理解。强烈建议你通过使用子标题和提纲使这份实验报告尽可能易读。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方案，或者理想情况下根据其性能、编写难度（例如，产生无错误实现所需的小时数）、可读性（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（或对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) 实施挑战。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，以及是什么最终帮助你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、不变式和前置条件的，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）。

   (e) 实践活动。包括对上述实践活动的回答和一些深思熟虑的评论。

4. 请同时填写作业所花费的小时数以及任何其他意见。

5. 请尽快让课程工作人员知道在实验课期间遇到的问题，或通过在 Ed 上发布问题。祝你好运！

6 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 post your test on EdStem (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 10, that's great too.)

6个额外加分项

对于改进测试套件，将给予额外加分。在 `tests` 目录（例如 `minnow/tests/recv connect.cc`）中的某个文件里添加一个测试用例，捕获一个真实且合理可能出现的、而现有测试套件尚未覆盖的 Bug。请将您的测试用例发布到 EdStem（公开即可）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 10 周您为 `Reassembler` 找到了一个良好的附加测试用例，那也很好。）