CS144：计算机网络导论 2025年秋季

实验检查点 3: TCP发送方

截止日期：10月19日（周日）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旧版本的解答。请在你的报告中完全披露任何合作者或任何灰色地带——坦诚是最好的策略。

0 概述

建议：在实现之前阅读整个实验文档。

在检查点0中，你实现了流控字节流（ByteStream）的抽象。在检查点1和2中，你实现了将不可靠数据报中携带的段转换为传入字节流的工具：Reassembler和TCPReceiver。现在，在检查点3中，你将实现连接的另一端。TCPSender是一个将出站字节流转换为将成为不可靠数据报有效载荷的段的工具。这将完成你对传输控制协议的实现（其实现可以说是世界上最普遍的计算机程序）。你将使用它与同学以及互联网上真正使用TCP的服务器对话。

1 开始

你的TCPSender实现将使用与检查点0-2中相同的Minnow库，并增加了额外的类和测试。开始步骤：

确保已提交检查点1的所有解决方案。请不要修改src目录顶层之外的任何文件，也不要修改webget.cc。否则你可能无法合并起始代码。

在实验作业仓库内，运行 `git fetch --all` 以获取实验作业的最新版本。

通过运行 `git merge origin/check3-startercode` 下载检查点3的起始代码。（如果你已将"origin"远程重命名为其他名称，你可能需要在此处使用不同的名称，例如 `git merge upstream/check3-startercode`。）

确保构建系统已正确设置：`cmake -S . -B build`

编译源代码：`cmake --build build`

打开并开始编辑 `writeups/check3.md` 文件。这是你实验报告的模板，将包含在你的提交中。

提醒：请在工作时在本地Git仓库中进行频繁的小提交。如果你需要帮助确保做对了，请向同学或教学人员寻求帮助。你可以使用git log命令查看你的Git历史。

2 检查点3：TCP发送方

TCP是一种协议，可通过不可靠数据报可靠地传递一对流控字节流（每个方向一个）。双方参与TCP连接，每一方都是对方的对等方。每个对等方同时充当"发送方"（发送自己的出站字节流）和"接收方"（接收传入字节流）。

本周，你将实现TCP的"发送方"部分，负责从ByteStream（由某个发送方应用程序创建和写入）中读取数据，并将流转换为一系列出站TCP段。在远程端，TCP接收方[1]将这些段（到达的那些——它们可能不会全部到达）转换回原始字节流，并将确认和窗口通告发送回发送方。

[1] 重要的是要记住，接收方可以是有效TCP接收方的任何实现——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在可能行为截然不同的端点之间建立通用语言。

你的TCPSender将负责：

跟踪接收方的窗口（接收传入的TCPReceiverMessage及其ackno和窗口大小）

在可能时填充窗口，通过从ByteStream读取、创建新的TCP段（包括SYN和FIN标志如果需要），并发送它们。发送方应持续发送段，直到窗口已满或出站ByteStream没有更多内容可发送。

跟踪哪些段已发送但尚未被接收方确认——我们称这些为"未完成"段

如果自发送以来经过了足够的时间且尚未被确认，则重新发送未完成的段

★我为什么要做这个？ 基本原则是发送接收方允许我们发送的任何内容（填充窗口），并持续重传直到接收方确认每个段。这称为"自动重传请求"（ARQ）。发送方将字节流分成段并发送它们，尽可能多地按照接收方的窗口允许。多亏了你上周的工作，我们知道远程TCP接收方可以重建字节流，只要它至少收到每个索引标记的字节一次——无论顺序如何。发送方的工作是确保接收方至少收到每个字节一次。

2.1 TCPSender如何知道段是否丢失？

你的TCPSender将发送一堆TCPSenderMessage。每个将包含来自出站ByteStream的（可能为空的）子串，用序列号索引以指示其在流中的位置，并在流的开头标记SYN标志，在结尾标记FIN标志。

除了发送这些段之外，TCPSender还必须跟踪其未完成的段，直到它们占用的序列号被完全确认为止。定期，TCPSender的所有者将调用TCPSender的tick方法，表示时间的流逝。TCPSender负责查看其未完成的TCPSenderMessage集合，并判断最早发送的段是否已未完成太久而没有得到确认（即其所有序列号都未被确认）。如果是，则需要重新传输（再次发送）。

以下是"未完成太久"的含义的规则[2]。你将实现这个逻辑，它有些细节化，但我们不希望你担心隐藏的测试用例试图绊倒你，或者把这当作SAT上的文字题。本周我们会给你一些合理的单元测试，并在Lab 4中完成整个TCP实现后提供更完整的集成测试。只要你100%通过那些测试且你的实现合理，就没问题。

[2] 这些基于TCP"真正"规则的简化版本：RFC 6298，建议5.1到5.6。这里的版本有些简化，但你的TCP实现仍然能够与互联网上的真实服务器通信。

★我为什么要做这个？ 总体目标是让发送方及时检测段何时丢失并需要重新发送。重新发送前等待的时间量很重要：你不希望发送方等待太久才重新发送段（因为那会延迟字节流向接收应用程序），但你也不希望它重新发送一个如果发送方再等一会儿就会被确认的段——那会浪费互联网宝贵的容量。

每隔几毫秒，你的TCPSender的 `tick` 方法将被调用，带有一个参数，告诉你自上次调用该方法以来经过了多少毫秒。使用这个来维护TCPSender存活的总毫秒数的概念。请不要尝试从操作系统或CPU调用任何"时间"或"时钟"函数——tick方法是你唯一获取时间流逝的途径。这保持了确定性和可测试性。

当TCPSender被构造时，它会收到一个参数，告知重传超时（RTO）的"初始值"。RTO是重新发送未完成TCP段之前等待的毫秒数。RTO的值会随时间变化，但"初始值"保持不变。起始代码将RTO的"初始值"保存在名为 `initial_RTO_ms` 的成员变量中。

你将实现重传计时器：一个可以在某个时间启动的闹钟，一旦RTO经过，闹钟就会响起（或"过期"）。我们强调，这个时间流逝的概念来自tick方法的调用——而不是获取实际时间。

每次发送包含数据（序列空间中非零长度）的段时（无论是第一次还是重传），如果计时器未运行，启动它使其在RTO毫秒后过期（对于当前的RTO值）。所谓"过期"，我们指的是时间将在未来某个毫秒数耗尽。

当所有未完成数据都被确认时，停止重传计时器。

如果 `tick` 被调用且重传计时器已过期：

(a) 重新发送最早（最低序列号）的尚未被TCP接收方完全确认的段。你需要将未完成的段存储在某个内部数据结构中以实现这一点。

(b) 如果窗口大小非零：

i. 跟踪连续重传次数，并递增它，因为你刚刚重传了某些内容。你的TCPConnection将使用此信息来判断连接是否无望（连续重传次数过多）并需要中止。

ii. 将RTO的值加倍。这称为"指数退避"——它减慢了在糟糕网络上的重传，以避免进一步堵塞网络。

(c) 重置重传计时器并启动它，使其在RTO毫秒后过期（考虑到你可能刚刚加倍了RTO的值！）。

当接收方给发送方一个确认收到新数据的ackno（ackno反映的绝对序列号大于任何先前的ackno）时：

(a) 将RTO设回其"初始值"。

(b) 如果发送方有任何未完成数据，重启重传计时器使其在RTO毫秒后过期（对于当前的RTO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，请将其添加到现有文件中（`tcp_sender.hh` 和 `tcp_sender.cc`）。

2.2 实现TCP发送方

好了！我们已经讨论了TCP发送方的基本思想（给定一个出站ByteStream，将其分成段，发送给接收方，如果它们没有及时被确认，则持续重新发送）。我们还讨论了何时可以得出结论认为未完成的段已丢失并需要重新发送。

现在是你将提供的具体接口的时候了。有四个重要事件需要处理：

`void push( const TransmitFunction& transmit );` — TCPSender被要求从出站字节流填充窗口：它从流中读取并发送尽可能多的TCPSenderMessage，只要有新字节可读且窗口中有可用空间。它通过调用提供的 `transmit()` 函数来发送它们。

你需要确保发送的每个TCPSenderMessage完全在接收方的窗口内。使每个消息尽可能大，但不超过 `TCPConfig::MAX_PAYLOAD_SIZE` 给出的值。

你可以使用 `TCPSenderMessage::sequence_length()` 方法来计算段占用的序列号总数。请记住SYN和FIN标志也各占用一个序列号，这意味着它们在窗口中占用空间。

★如果窗口大小为零我该怎么办？ 如果接收方通告了零大小的窗口，push方法应该假装窗口大小为1。发送方最终可能会发送一个被接收方拒绝（且未确认）的单字节，但这也可以刺激接收方发送一个新的确认段，揭示其窗口中已有更多空间打开。没有这个，发送方永远不会知道它被允许重新开始发送。

这是你的实现对于零大小窗口情况应该具有的唯一特殊行为。TCPSender不应该实际记住一个虚假的窗口大小1。特殊情况仅在push方法内部。另外请注意，即使窗口大小为1（或20，或200），窗口仍然可能是满的。"满"窗口不等同于"零大小"窗口。

`void receive( const TCPReceiverMessage& msg );` — 从接收方收到一条消息，传达了窗口的新左边缘（= ackno）和右边缘（= ackno + 窗口大小）。TCPSender应查看其未完成段的集合，并移除任何现在已被完全确认的段（ackno大于段中的所有序列号）。

`void tick( uint64_t ms_since_last_tick, const TransmitFunction& transmit );` — 时间已过——自上次调用此方法以来经过了一定数量的毫秒。发送方可能需要重传一个未完成的段；它可以调用 `transmit()` 函数来执行此操作。（提醒：请不要在你的代码中使用现实世界的"时钟"或"gettimeofday"函数；唯一的时间流逝参考来自 `ms_since_last_tick` 参数。）

`TCPSenderMessage make_empty_message() const;` — TCPSender应生成并发送一个零长度消息，序列号设置正确。如果对端想要发送一个TCPReceiverMessage（例如因为它需要确认来自对端发送方的某些内容）并且需要生成一个TCPSenderMessage来配合它，这很有用。注意：这样的不占用序列号的段不需要被跟踪为"未完成"，也不会被重传。

要完成检查点3，请查看 `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。我们预计你需要添加私有方法和成员变量，以及可能的辅助类。

2.3 常见问题和特殊情况

我的TCPSender在receive方法告知之前应该假设接收方的窗口大小为多少？ 一。

如果确认只部分确认了某个未完成段怎么办？我应该尝试裁剪已确认的字节吗？ TCP发送方可以这样做，但在本课程中，不需要搞得太复杂。将每个段视为完全未完成，直到它被完全确认——它占用的所有序列号都小于ackno。

如果我发送了三个单独的段包含“a”、“b”和“c”，且它们从未被确认，我以后可以在一个包含“abc”的大段中重传它们吗？还是必须单独重传每个段？ 同样：TCP发送方可以这样做，但在本课程中，不需要搞得太复杂。只需单独跟踪每个未完成的段，当重传计时器过期时，再次发送最早的未完成段。

我应该在我的“未完成”数据结构中存储空段并在必要时重传它们吗？ 不——唯一应该被跟踪为未完成并可能被重传的段是那些传递了一些数据的段——即消耗了序列空间中某些长度的段。不占用序列号的段（没有SYN、有效载荷或FIN）不需要被记住或重传。

如果在此PDF发布后有更多常见问题，我在哪里可以阅读？
请定期查看网站（https://cs144.github.io/lab_faq.html）和Ed。

3 开发与调试建议

在 `tcp_sender.cc` 文件中实现TCPSender的公共接口。你可以在 `tcp_sender.hh` 中的TCPSender类中添加任何你喜欢的私有成员。

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

请重新阅读检查点0文档中关于"使用Git"的部分，并记住将代码保存在分发的Git仓库的main分支上。进行小的提交，使用好的提交消息。

请努力使你的代码对评分的CA可读。使用合理且清晰的变量命名约定。使用注释来解释复杂或微妙的代码段。使用"防御性编程"。在你的设计中使用模块化。重复的代码块和巨大的函数会使你的代码难以理解。

4 动手实践活动

恭喜你——你已经完成了传输控制协议的完整工作实现，其实现可以说是地球上最普遍的计算机程序。是时候庆祝一下了！你将与Linux的TCP以及实验伙伴通信，然后修改你的webget（来自检查点0）以使用你的TCP实现。在你的报告中，描述你做了什么，回答以下问题，并尝试找到一些有趣的东西来讨论！

4.1 在你自己的VM内进行实验

我们给了你一个客户端程序（`./build/apps/tcp_ipv4`），它使用你的TCPSender和TCPReceiver通过互联网讲TCP-over-IP[3]。我们还给了你一个类似的程序（`./build/apps/tcp_native`），它使用Linux TCPSocket。

[3] 如果你好奇这个程序如何工作，`tcp_peer.hh` 和 `tcp_over_ip.cc` 文件可能是我们将你的TCPSender/TCPReceiver粘合为一致的TCP对等方的有趣部分。

大问题：你的TCP实现（tcp_ipv4）能与Linux的TCP（tcp_native）互操作吗？

4.1.1 让Linux的TCP与自己对话

首先，让我们确保Linux的TCP实现可以与自己对话。在端口9090上将Linux的TCP作为"服务器"运行。在你的VM上运行：`./build/apps/tcp_native -l 0 9090`

接下来，尝试将Linux的TCP用作"客户端"：发起连接并向服务器发送第一个SYN段的对等方。在你的VM上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 from 169.254.144.1:36568` 的内容，"客户端"将打印类似 `DEBUG: Connecting to 169.254.144.1:9090... DEBUG: Successfully connected to 169.254.144.1:9090` 的内容。

尝试在每个窗口中打字，你会在另一个窗口上看到相同的字节。

要结束流，请按 `ctrl-D`（在空行上）关闭该方向上的ByteStream Writer。如果一切顺利，你将在你按ctrl-D的终端上看到 `Outbound stream...finished`，在另一个终端上看到 `Inbound stream...finished`。注意另一个对等方可以继续向"已关闭"的对等方发送——流的每个方向可以独立关闭，而不会阻止另一个方向继续。

现在在另一个终端中按 `ctrl-D` 来结束第二个方向的流。如果一切顺利，两个程序将退出并在两个终端中返回命令行。这表明TCP连接已在两个方向上完成。

4.1.2 让你的TCP与Linux的对话

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` 以授予你的实现发送原始互联网数据报的权限而无需root权限。每次重启VM时都需要重新运行此命令。

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`（这是你的TCP实现）。连接是否像之前一样建立，每个对等方是否仍然可以在另一个对等方的窗口上打字并显示文本？如果是，给自己一个赞——你已经赢得了它！如果不是...是时候开始调试了。你可以用类似以下命令捕获TCP段：`sudo rm -f /tmp/capture.raw; sudo tcpdump -n -w /tmp/capture.raw -i tun144 --print --packet-buffered`；生成的 `/tmp/capture.raw` 文件可以像以前一样在wireshark中可视化。

在你每个方向上打了一些字之后，尝试关闭其中一个ByteStream并继续在另一个方向上打一些字。两个程序在两个流都通过ctrl-D结束后是否都能干净地退出？它们应该——虽然你可能需要看到tcp_ipv4稍等一下以减少"两将军问题"的机会。它什么时候需要等待（当它是第一个关闭的还是第二个关闭的）？这是否与课堂上讨论的内容匹配？

4.1.3 尝试通过"一兆字节挑战"

一旦看起来你可以进行基本对话，尝试在 `tcp_ipv4`（你的TCP）和 `tcp_native`（Linux的TCP）之间发送文件。

创建一个12345字节的随机文件作为"/tmp/big.txt"：

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

你可以选择传输方向——即客户端还是服务器是发送文件的那个。

让服务器在接受传入连接后立即发送文件，将标准输入重定向为从文件读取：

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

让客户端接收文件，通过从/dev/null重定向关闭其出站流，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
```

比较两个文件以确保它们相同：

```
sha256sum /tmp/big.txt
sha256sum /tmp/big-received.txt
```

如果SHA-256哈希值匹配，你几乎可以确定文件传输正确。尝试用一个小文件（12字节），然后65534字节（略小于2^{16}），然后65537字节（略大于2^{16}），然后200000字节，然后完整的一兆字节（1000000字节）。如果它们都匹配，给自己一个更大的赞！如果不匹配...是时候调试了（可能用tcpdump和wireshark如上所述）。

4.2 与朋友对话

如果以上都能正常工作，尝试通过互联网与实验伙伴通信！你们中的一个将如上运行 `tcp_native` 作为服务器。另一个将运行 `tcp_ipv4` 作为客户端，连接到实验伙伴在CS144专用网络上的地址（10.144...）。

你们能互相打字并成功干净地结束两个流吗？如果可以，你能通过一兆字节挑战吗（通过互联网向你的实验伙伴的VM成功发送一个随机的1000000字节文件，两边的SHA-256哈希值完全匹配）？如果可以，恭喜...现在交换角色并尝试在另一个方向发送文件！

你有耐心成功发送给实验伙伴的最大文件是多少？在你的实验报告中，包括两个文件的大小（发送方 `ls -l /tmp/big.txt` 和接收方 `ls -l /tmp/big-received.txt` 的输出）以及 `sha256sum` 的结果。

4.3 重新审视webget

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）。我们希望你将其切换为使用你自己的TCP实现，而不更改其他任何内容。我们认为你需要做的只是：

将 #include "socket.hh" 替换为 #include "tcp_minnow_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

★我为什么要做这个？ 通常Linux内核会在用户进程退出后继续处理TCP连接的"干净关闭"（并释放其端口预留）。但因为你的TCP实现全部在用户空间中，除了你的程序之外没有其他东西来跟踪连接状态。添加此调用使套接字等待直到连接完全关闭。

重新编译，然后运行 `make check_webget` 确认你已经完成了完整的循环：你在自己完整的TCP"栈"之上编写了一个基本的Web获取器，它仍然成功地与真正的Web服务器通信。如果遇到问题，尝试手动运行程序：`./build/apps/webget cs144.keithw.org /hasher/xyzzy`。你将在终端上获得一些可能有帮助的调试输出。

5 提交

在你的提交中，请仅更改 `src` 目录（和 `apps/webget.cc`）中的 `.hh` 和 `.cc` 文件。在这些文件中，请随意添加必要的私有成员，但请不要更改任何类的公共接口。

在提交任何作业之前，请按顺序运行以下命令：

(a) 确保已将所有更改提交到Git仓库。记住：在编码时进行小的提交。

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

(c) `cmake --build build --target check3`（确保自动化测试通过）

(d) 可选：`cmake --build build --target tidy`

在 `writeups/check3.md` 中撰写报告。此文件应为大约20到50行的文档，每行不超过80个字符。报告应包含以下部分：

(a) 程序结构与设计。(b) 替代设计选择。(c) 实现挑战。(d) 残余错误。(e) 动手实践活动——包括对上述问题的回答和一些有思考的评论。

请同时填写完成作业所花的小时数和任何其他评论。

如果有任何问题，请尽快通知课程工作人员。祝你好运！

6 额外学分

对测试套件的改进将获得额外学分。添加一个测试用例，能捕获现有测试套件尚未捕获的真实错误。请将你的测试发布在EdStem上，以便我们查看并决定是否将其添加到整体测试套件中。