CS144实验记录

CheckPoint 0

1. 配置GNU/Linux环境

这门课的实验要求使用虚拟机来模拟GNU/Linux或者直接安装Ubuntu系统。很不巧，我使用的是M1芯片的Macbook air，不管是安装虚拟机或者装双系统或者使用docker来获得Ubuntu container都会对性能造成很大的影响。因此我使用Github Codespace，正好blank模版提供的也是Ubuntu 20.04.6的环境。

```
@zzr997good ➜ /workspaces/codespaces-blank $ cat /etc/os-release
NAME="Ubuntu"
VERSION="20.04.6 LTS (Focal Fossa)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 20.04.6 LTS"
VERSION_ID="20.04"
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
VERSION_CODENAME=focal
```

接下来按照要求的安装对应的package

```
sudo apt update && sudo apt install git cmake gdb build-essential clang \
clang-tidy clang-format gcc-doc pkg-config glibc-doc tcpdump tshark
```

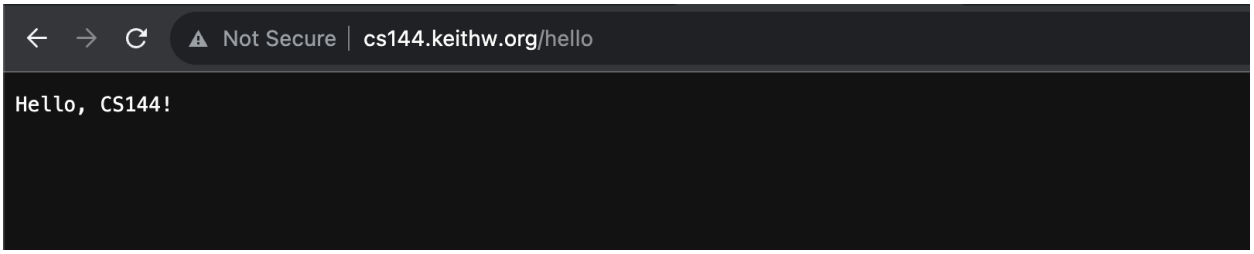
到此，环境配置完成。

2. 玩一玩networking

2.1 Fetch一个网页

要求是让你尝试直接用浏览器去访问一个网页和用telnet去发送HTTP REQUEST去访问一个网页，观察两者有什么不同。

直接用浏览器去访问： http://cs144.keithw.org/hello，可以看出所有的发送请求的过程已经被浏览器包装起来了。作为一个client，唯一需要知道的就是网页的地址。



使用telnet去访问：没有telnet工具的可以先安装一下telnet。然后需要建立和服务器的连接，然后发送HTTP Get Request，request中包含需要访问的URL路径和Host名称。然后关闭连接等待回复。

```
sudo apt install telnet
```

```
@zzr997good ➜ /workspaces/codespaces-blank $ telnet cs144.keithw.org http
Trying 104.196.238.229...
Connected to cs144.keithw.org.
Escape character is '^]'.
GET /hello HTTP/1.1
Host: cs144.keithw.org
Connection: close

#Response
HTTP/1.1 200 OK
Date: Mon, 27 Nov 2023 05:24:02 GMT
Server: Apache
Last-Modified: Thu, 13 Dec 2018 15:45:29 GMT
ETag: "e-57ce93446cb64"
Accept-Ranges: bytes
Content-Length: 14
Connection: close
Content-Type: text/plain

Hello, CS144!
Connection closed by foreign host.
```

2.2 调用stanford的smtp服务来给自己发送邮箱

我做不了，我不是Stanford的学生，我没有sunetid，我彻底失败。

2.3 双工通信

用netcat和telnet进行一个双向通信，就类似是两个人在一个聊天室聊天。你发送的信息会被echo在当前聊天室。

```
#terminal A
@zzr997good ➜ /workspaces/codespaces-blank $ netcat -v -l -p 9090
Listening on 0.0.0.0 9090
Connection received on localhost 39302
Hello there, here is A
Hello there, here is B
```

```
#terminal B
@zzr997good ➜ /workspaces/codespaces-blank $ telnet localhost 9090
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello there, here is A
Hello there, here is B
```

3. 用OS stream socket写一个网络程序去fetch一个网页

3.1 拉源代码

```
git clone https://github.com/cs144/minnow
```

想要把修改完的代码上传到自己仓库的话记得在codespace里面添加新的ssh：
https://docs.github.com/en/authentication/connecting-to-github-with-ssh/about-ssh

看样子整个项目都是用CMake进行build的，但是codespace本身自带的CMake的版本比较低，而项目要求3.24.2版本的CMake，因此我们需要重新安装一个CMake

```
#卸载旧版本CMake
sudo apt-get remove cmake
#去官网下载一个新版本的CMake的Linux x86-64的binary
#解压
tar -zxvf cmake-3.26.5-linux-x86_64.tar.gz
#安装
sudo mv cmake-3.26.5-linux-x86_64 /opt/cmake-3.26.5
#添加环境变量
vim ~/.bashrc
#export PATH=/opt/cmake-3.26.5/bin:$PATH
cmake --version
```

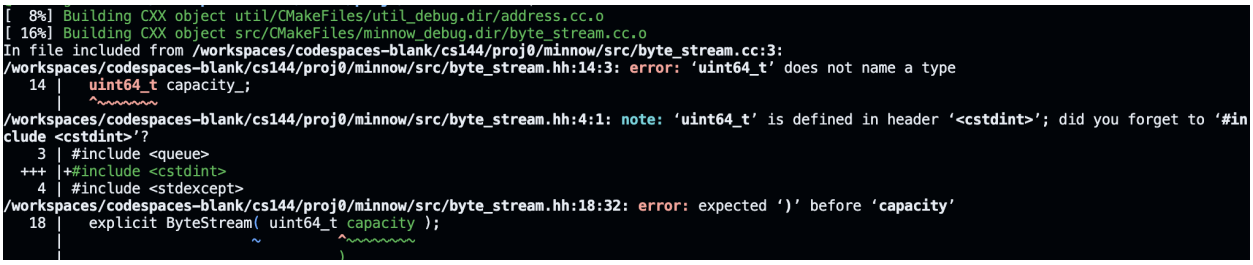
此时CMake已经是3.26.5版本的了，够用了

开始先编译build一遍源代码

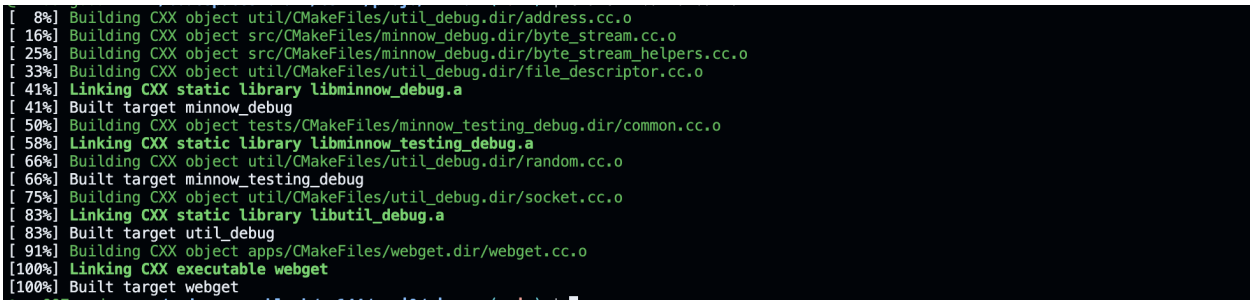
```
cd minnow/
cmake -S . -B build
cmake --build build
#报错
```

我排查了一下问题，是我的gcc和g++的版本太低了。

按照教程https://www.ovenproof-linux.com/2016/09/upgrade-gcc-and-g-in-ubuntu.html 把gcc和g++升级到了13。重新编译build一下，出现了以下错误。



全是关于uint64_t的，去TMD的，直接暴力往byte_stream.hh里面添加头文件<cstdint>，问题解决。



3.2 看看modern C++圣经

3.3 读一下file_descriptor提供的API

3.4 实现get_URL

其实就是用已经封装好的TCPSocket类来和服务器进行连接，然后发送按照格式发送request

```
void get_URL( const string& host, const string& path )
{
  // cerr << "Function called: get_URL(" << host << ", " << path << ")\n";
  // cerr << "Warning: get_URL() has not been implemented yet.\n";
  TCPSocket sock;
  Address server(host,"http");
```

```
  // build connection with the server
  sock.connect(server);
  // send request to the server
  string requst="GET "+path+" HTTP/1.1\r\n"+"Host: "+host+"\r\n"+"Connection: close\r\n"+"\r\n";
  sock.write(requst);
  // receive response from the server
  string response;
  while(!sock.eof()){
    sock.read(response);
    cout<<response;
  }
  sock.close();
}
```

测试一下

```
rm -rf build/
cmake -S . -B build
cmake --build build
cmake --build build --target check_webget
```

测试结果都通过

```
@zzr997good → .../codespaces-blank/cs144/proj0/minnow (main) $ cmake --build build --target check_webget
Test project /workspaces/codespaces-blank/cs144/proj0/minnow/build
    Start 1: compile with bug-checkers
1/2 Test #1: compile with bug-checkers ........   Passed    0.56 sec
    Start 2: t_webget
2/2 Test #2: t_webget .........................   Passed    0.73 sec
```

4. 设计一个bytestream类

要求设计一个单线程的在本地的bytestream类，有两个继承类Writer和Reader。其实就是一个生产者消费者模型，同时不用考虑同步问题。bytestream有一个capacity，表示该byetestream的buffer中最多能够在内存中占用多少空间。Writer写的时候，最多能够把buffer占满，剩下的数据就自动舍弃。(这一点很迷)。Reader读的时候可以从中取出一定长度的字节并放到自己的buffer里去。由于是在本地的bytestream，因此**可以把writer想象成TCP的buffer，receiver想象成application获取数据buffer。**

bytestream的buffer我觉得主要有以下几种实现方法

1. 使用queue：这是最显而易见的，因为它头文件自动帮你加了<queue>，但是因为C++的queue本身是可扩容的，用capacity去限制size需要特别注意push的时候不能超过capacity。因此push的时候push的有效长度应该为min(capacity-que.size(),data.size())
2. 使用vector：使用vector去模拟一个queue也是很常见的做法。push的时候不断push_back，pop的时候就用一个指针去维护当前stream的头部位置。不过这种实现方法太占内存，vector会一直扩容。
3. 用vector去模拟一个循环队列：固定vector的size是capacity，然后用两个指针去模拟队列头尾。每次移动都进行取模操作。

我用循环队列去实现。结果实现下来发现peek()真的很难用循环队列去返回一个string_view。

```
string_view Reader::peek() const
{
  // Your code here.
  if(is_finished()){
    throw runtime_error("Try to peek from finished stream");
  }
  if(has_error()){
    throw runtime_error("Try to peek from errored stream");
  }
  if(bytes_buffered_==0){
    return "";
  }
  size_t head = bytes_popped_ % capacity_;
  string ret="";
  for(size_t i=0;i<bytes_buffered_;i++){
    ret+=buffer_[head];
    head=(head+1)%capacity_;
  }
  return ret;
}
```

因为最后返回的bytes可能是是循环队列的尾部和头部连接，没办法直接返回string_view。如果临时产生一个string，返回的过程中string ret被销毁，string_view直接没有意义。于是我灰溜溜的改用queue去做buffer_了，如果用string去作为buffer_也可以，但是pop的时候需要用substr去产生新的buffer_比较麻烦。

但是当我用queue<char>去实现的时候，我发现第八个测试死活都会出现stack overflow的错误，排查了很久才知道原因。因为queue底层是使用deque实现的，而deque是当内存超过512B的时候就会分配一块新的512B的block，然后在指针数组中添加一个新的指针指向新的block的首地址。**因此queue<char>的内存超过512次push后并不连续，在返回string_view的时候就会出现stack overflow的错误。因为string_view需要一块连续内存的视图。**

最后我用string作为buffer_实现，每次pop的时候生成一个新的string。

```
string_view Reader::peek() const
{
  // Your code here.
  if(is_finished()){
    return string_view();
  }
  if(has_error()){
    return string_view();
  }
  if(buffer_.empty()){
    return string_view();
  }
  return string_view(buffer_);
}
```

这个peek的API真的恶心到我了，为了追求效率一定要返回string_view。

5. 提交代码

提交之前有以下几点要确认：

1. 测试全都通过了
2. 代码格式已经规范化了

```
cmake --build build --target check0
cmake --build build --target format
```

我在运行format用clang format规范代码格式的时候发现最后一行PackConstructorInitializers: NextLine无法识别。肯定又是clang的版本太低导致的，于是我更新了clang的版本

```
#下载安装clang18
wget https://apt.llvm.org/llvm.sh
chmod +x llvm.sh
sudo ./llvm.sh 18
sudo apt install clang-format-18
#切换clang版本
sudo update-alternatives --install /usr/bin/clang-format clang-format /usr/bin/clang-format-18 100
```

然后再次规范化代码即可。

CheckPoint 1

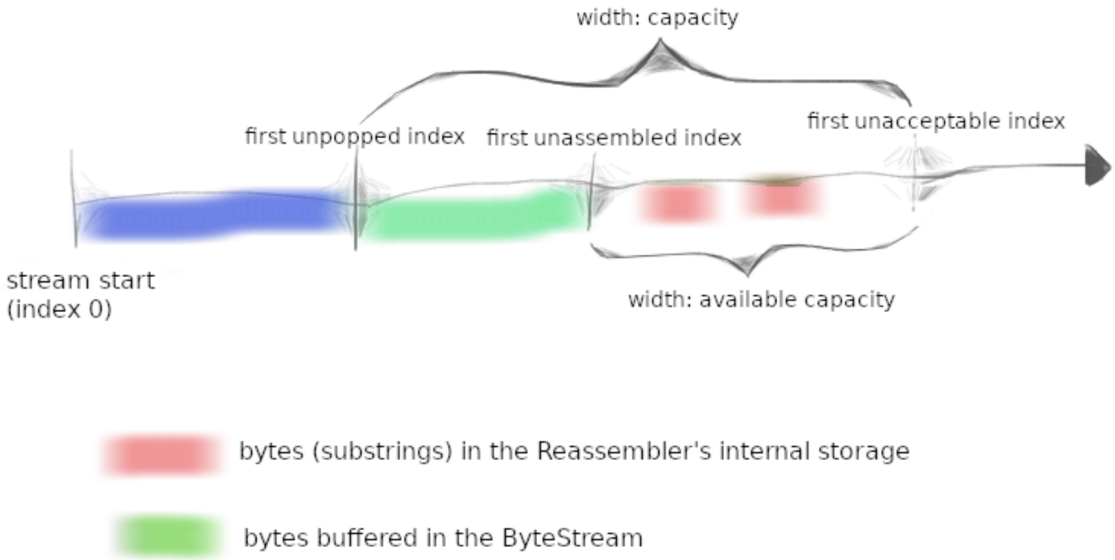
1. 拉源码

```
git fetch
git merge origin/check1-startercode
cmake -S . -B build
cmake --build build
```

2. 实现一个Reassembler

这次要求实现一个Reassembler，这样receiver可以使用这个Reassembler去将由IP传来的datagram进行重组和去重，TCP的reliable也由此实现。

2.1 实现原理



其实就是一个滑动窗口，对于receiver而言，数据流动的方向为，IP→Reassembler→Writer ByteStream→Application。蓝色部分表示已经发送给application的数据，因此不需要进行任何的存储。绿色部分表示由Reassembler排好的紧接在已经发送好的数据后的连续的数据，这一部分数据可以随时发送给application，因此缓存在Writer Bytestream中，等待application读取。红色部分表示提前发来的在capacity(window) 之内的数据，由于前面还缺少一些数据让他们成为有序的连续数据，因此可以先缓存在Reassembler中，等待缺失的数据到了以后再联合成有序的连续数据发送给Writer。超出capacity(first unacceptable index)的部分直接抛弃掉。

这次提供的接口主要就两个

```
void insert( uint64_t first_index, std::string data, bool is_last_substring, Writer& output );
uint64_t bytes_pending() const;
```

然后我的实现方法就是在Reassembler类里面维护一些私有数据成员还有两个私有函数成员

```
private:
  struct segment
  {
    uint64_t first_index;
    uint64_t len;
    std::string data;
    bool operator<( const segment& other ) const
    {
      return first_index == other.first_index ? len < other.len : first_index < other.first_index;
    }
  };
  std::set<segment> unassembled_segments;
  uint64_t first_unassembled_index;
  uint64_t bytes_pended;
  bool got_last_substring;
```

```
std::set<segment>::iterator try_merge( std::set<segment>::iterator cur_data );
std::set<segment>::iterator merge_overlapped( std::set<segment>::iterator prev_seg,
                                              std::set<segment>::iterator cur_data );
```

1. segment数据结构表示在Reassembler buffer里面存储的离散的字节流。
2. unassembled_segments就相当于是Reassembler buffer
3. first_unassembled_index就是如上图所示的第一个不连续的index，其值应该等于Writer.bytes_pushed()
4. bytes_pended表示在Reassembler buffer存储的所有离散的bytes的总长度
5. got_last_substring表示Reassembler是否已经接收到了last substring
6. try_merge()和merge_overlapped()是在insert()的时候用来尝试将Reassembler buffer中可以合并的bytes合并成连续区间的
7. 由于insert的时候，提供了output，因此first_unacceptable_index可以通过first_unassembled_index+output.available_capacity()计算出来
8. first_unpopped_index其实就是Reader.bytes_popped()，不过这个是由application的读取速度决定的，当application读取的快，窗口就移动的快，否则窗口就移动的慢。很多字节都变成了unacceptable
9. 整个capacity其实就是output的capacity，只不过从first_unpopped_index到first_unassembled_index那部分被buffer在了Writer的buffer中，而available capacity不仅仅是Writer的剩余capacity，也是Reassembler buffer的整个capacity。
10. 新来的数据首先要进行切割，保证其开头index不会小于first_unassembled_index，结尾不会大于first_unacceptable_index，这样可以保证Reassembler buffer和Writer的buffer(如果成功push到writer之后)也不会溢出
11. 接着对切割后的数据进行尝试merge，这是一个递归的过程，如果该数据无法和前面也无法和后面的数据进行merge，那么说明该数据是离散数据。无法merge，直接存入set。否则要让其和前后的数据merge得到新的数据，然后将新的数据继续和前后的数据进行merge，直到无法merge。
12. 判断merge之后的数据start_index是否为first_unassembled_index，如果是的话，将其从set中弹出，直接push到output的buffer中。
13. 在merge和弹出的过程中不断对bytes_pended进行加减操作。

具体实现代码请看代码仓库。

2.2 FAQ

3. 如何在测试的过程中debug

在CheckPoint0里面我debug都是凭感觉debug，也没用gdb，也磕磕绊绊过了。这次test比较多，感觉实在不方便，就用了vscode自带的gdb插件进行debug。毕竟有UI，又可以方便地打断点，还可以动态

监视所有的变量。好用！

由于我是在codespace里面开发的，所以主要就是在/workspaces/codespaces-blank就是我的current working directory，然后minnow就在cwd下面，因此只需要在cwd里面创建一个.vscode文件夹(作为working directory的子目录)，然后在.vscode下面添加launch.json和task.json就可以随便打开/workspaces/codespaces-blank/minnow/tests/下任何一个测试文件打断点进行debug了。具体可以参考https://segmentfault.com/a/1190000039087458

4. Submit

还是老样子

1. 检查test过了没

```
Test project /workspaces/codespaces-blank/minnow/build
      Start  1: compile with bug-checkers
 1/17 Test  #1: compile with bug-checkers ........   Passed    4.67 sec
      Start  3: byte_stream_basics
 2/17 Test  #3: byte_stream_basics ...............   Passed    0.01 sec
      Start  4: byte_stream_capacity
 3/17 Test  #4: byte_stream_capacity .............   Passed    0.01 sec
      Start  5: byte_stream_one_write
 4/17 Test  #5: byte_stream_one_write ............   Passed    0.01 sec
      Start  6: byte_stream_two_writes
 5/17 Test  #6: byte_stream_two_writes ...........   Passed    0.01 sec
      Start  7: byte_stream_many_writes
 6/17 Test  #7: byte_stream_many_writes ..........   Passed    0.05 sec
      Start  8: byte_stream_stress_test
 7/17 Test  #8: byte_stream_stress_test ..........   Passed    0.04 sec
      Start  9: reassembler_single
 8/17 Test  #9: reassembler_single ...............   Passed    0.01 sec
      Start 10: reassembler_cap
 9/17 Test #10: reassembler_cap ..................   Passed    0.01 sec
      Start 11: reassembler_seq
10/17 Test #11: reassembler_seq ..................   Passed    0.02 sec
      Start 12: reassembler_dup
11/17 Test #12: reassembler_dup ..................   Passed    0.04 sec
      Start 13: reassembler_holes
12/17 Test #13: reassembler_holes ................   Passed    0.01 sec
      Start 14: reassembler_overlapping
13/17 Test #14: reassembler_overlapping ..........   Passed    0.01 sec
      Start 15: reassembler_win
14/17 Test #15: reassembler_win ..................   Passed    0.49 sec
      Start 16: compile with optimization
15/17 Test #16: compile with optimization ........   Passed    0.14 sec
      Start 17: byte_stream_speed_test
            ByteStream throughput: 1.08 Gbit/s
16/17 Test #17: byte_stream_speed_test ...........   Passed    0.15 sec
      Start 18: reassembler_speed_test
            Reassembler throughput: 9.44 Gbit/s
17/17 Test #18: reassembler_speed_test ...........   Passed    0.15 sec
```

2. 格式化代码

```
cmake --build build --target format
```

3. commit并提交代码到仓库

CheckPoint 2

0. 背景故事

一个TCPReceiver，主要负责以下两个工作：

1. 接收从TCPSender发来的消息，把data放进reassembler。
2. 发送ack包，包中包含ackno和windowsize。

1. 拉源代码

老样子

```
git fetch --all
git merge origin/check2-startercode
git merge upstream/check2-startercode
cmake -S . -B build
cmake --build build
```

2. 实现TCPReceiver

这次要实现的TCPReceiver就是一个很简单的Receiver，他不会发送任何数据包给TCPSender，只是根据收到的packet来发送ack包。

2.1 实现64bit index到32bit seqno的转化

由于CheckPoint1和CheckPoint0里的bytestream index都是64位的，但在tcp header里只有32位用来存放seqno，因此我们需要用32位循环的seqno来表示64位的bytestream index。源代码中已经将uint32_t封装成了Wrap32类，其中包含两个方法：

```
static Wrap32 Wrap32::wrap( uint64_t n, Wrap32 zero_point )
uint64 t unwrap( Wrap32 zero point, uint64_t checkpoint ) const
```

其中静态wrap方法使用来将一个64位的Absolute Sequence Numbers(从0开始的seqno)转换成32位的seqno，同时需要提供一个zero_point，这是TCP三次握手时Sender发送的随机生成的SYN的seqno，也就是ISN。而方法unwrap就来将当前Wrap32对象转换成最靠近checkpoint的Absolute Sequence Numbers。也就是说Wrap32提供了squno和Absolute Sequence Numbers之间的转换。至于Absolute Sequence Numbers和bytestream index之间的转换非常简单。具体可以参考下图。

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:

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"

也就是说bytestream index是不包括SYN和FIN的仅含data的0-index，也就是Reassembler和Writer中使用的index，而Absolute Sequence Numbers是包含SYN，FIN和data的0-index。至于seqno则是根据一个随机产生的ISN将Absolute Sequence Numbers转换成32位循环形式的包含在返回包中的index。

接下来实现这两个API。

```
Wrap32 Wrap32::wrap( uint64_t n, Wrap32 zero_point )
{
  return zero_point + static_cast<uint32_t>( n );
}

uint64_t Wrap32::unwrap( Wrap32 zero_point, uint64_t checkpoint ) const
{
  // Step1: wrap the checkpoint and find the offset in uint32_t range
  uint32_t offset = raw_value_ - wrap( checkpoint, zero_point ).raw_value_;
  // Step2: get the potential result
  uint64_t result = checkpoint + offset;
  // Why this is a potential result?
  // 1. rwo_value_ could be smaller than wrapped checkpoint and in that case offset is wrap to a bigger dis
  // 2. offset could be bigger than 2^31 which cause offset-2^32 is a closer result
  if ( offset > ( 1u << 31 ) && result >= ( 1ull << 32 ) ) {
    result -= ( 1ull << 32 );
  }
  return result;
}
```

wrap的实现原理较为简单，直接将Absolute Sequence Numbers静态cast成32位，这和%(1<<32)的效果一样，得到在32位下的偏移量，然后将偏移量加上zero_point。

unwrap的实现比较tricky。因为不管是在64位数轴上还是32位数轴上，两个数字之间的偏移量是不变的。因此我们首先计算转换之后的checkpoint和当前raw_value_在32位数轴上的偏移量。接着将其加到64位checkpoint上，就得到了还原之后可能的结果。为什么说只是可能的结果呢？因为有以下几种情况：

1. 转换之后的checkpoint比当前raw_value_大，计算出来的偏移量因为也是uint32_t的，会从一个负数k变成2^32-k，这就导致还原之后得到了一个比checkpoint大并且较远的数字。
2. raw_value_比转换之后的checkpoint大，但是偏移量大于2^31，也就是整个区间的一半。那么得到的结果虽然是正常unwrap的结果，但是前面有一个离checkpoint更近的。

因此当offset>(1<<31)并且得到的结果比(1<<32)大时，可以减去一个区间长度，得到一个离checkpoint更近的结果。

实现完之后可以单独进行测试，如果按照材料所说用以下这行命令进行测试肯定无法通过，因为他单独加了一些testcase。

```
cmake --build build --target check2
```

所以我搜了一下单独进行测试用例的办法：

```
#首先列出所有的测试用例
cmake --build build --target help
#然后用ctest --test-dir指定build目录
#ctest -R 指定运行的测试
ctest --test-dir build -R wrapping_integers_wrap
ctest --test-dir build -R wrapping_integers_unwrap
```

2.2 实现TCPReceiver类

```
struct TCPSenderMessage
{
  Wrap32 seqno { 0 };
  bool SYN { false };
  Buffer payload {};
  bool FIN { false };

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

TCPSender发来的消息主要包含以下几个数据：

1. 是不是SYN信号
2. 是不是FIN信号
3. 包含的data payload

4. 发送的seqno

如果发送的包含SYN信号，那么seqno是由sender随机生成的ISN，也就是SYN信号的seqno。如果不包含SYN信号，并且payload不为空，那么seqno是payload第一位byte的seqno。如果仅包含FIN信号，那么seqno是FIN的seqno。也就是说SYN和FIN都会占用一位seqno。

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

TCPReceiver收到Sender发来的消息后产生的信息只包含两个数据：

1. 回复的ackno
2. windowsize来进行flow control

```
class TCPReceiver
{
public:
  /* The TCPReceiver receives TCPSenderMessages, inserting their payload
   * into the Reassemble at the correct stream index. */
  void receive( TCPSenderMessage message, Reassembler& reassembler, Writer& inbound_stream );

  /* The TCPReceiver sends TCPReceiverMessages back to the TCPSender. */
  TCPReceiverMessage send( const Writer& inbound_stream ) const;
};
```

TCPReceiver要实现的两个API已经提及过了。

实现思路：

1. 首先TCPRecerver中要保存一个私有数据成员用来表示是否收到了SYN信号

```
std::optional<Wrap32> zero_point { std::nullopt };
```

2. receive API的实现：
    a. 首先检查zero_point是否有值并且当前message是否包含SYN，如果都没有，那就直接返回，因为连接还没建立。
    b. 如果当前message包含SYN，说明这是SYN包(也可能包含data和FIN信号)，将zero_point赋值为message中包含的ISN(seqno)
    c. 计算message的bytestream index：方法是先将seqno逆转换成Absolute Sequence Numbers，然后根据Absolute Sequence Numbers和bytestream index的关系来计算bytestream index。那

么逆转换的checkpoint是多少呢？材料要求用reassembler中的first unassembled index来作为checkpoint，也就是bytes_pushed()。但是bytes_pushed()返回的是bytestream index而不是Absolute Sequence Numbers，二者的含义是不同的，因此还需要加上SYN在Absolute Sequence Numbers中占的一位，因此checkpoint是1+bytes_pushed()。

d. 计算bytestream index：如果该message中包含SYN，那么计算得到的Absolute Sequence Numbers必然为0，此时包含payload的bytestream index也必然是从0开始。如果不包含SYN，那么Absolute Sequence Numbers是包含了SYN的index，因此需要讲得到的Absolute Sequence Numbers-1才是最终的bytestream index。

e. 得到bytestream index后就可以用以下的API来向reassembler发送数据。

```
reassembler.insert( first_index, message.payload, message.FIN, inbound_stream );
```

3. send API的实现

a. ackno的填写：主要是把当前收到的所有数据量wrap成一个Wrap32类型。如果当前已经收到了SYN(zero_point有值)，那么SYN要占一位。如果当前已经收到了FIN(inbound_stream.is_closed())，那么FIN要占一位。另外所有的data_pushed()也需要占位，把以上三者考虑起来就得到了当前接收到的字节量，也就是期待的下一个byte的Absolute Sequence Numbers。那么将其根据zero_point包装成wrap32类型的数据并赋值就行。

b. window size的填写：注意window size是16位的，而inbound_stream.available_capacity()是32位的，因此需要截断。如果inbound_stream.available_capacity()>0xFFFF，那么window size最大也就只能是0xFFFF。

3. 测试

老样子gdb慢慢调试，这次他添加了一些对reassembler的测试用例，需要因此我也改动了reassembler的一个小地方，也就是完全overlapping的数据如果已经在set中，那么就insert就不用继续考虑merge了。

```
      Start 14: reassembler_overlapping
13/29 Test #14: reassembler_overlapping ..........   Passed    0.02 sec
      Start 15: reassembler_win
14/29 Test #15: reassembler_win ..................   Passed    0.48 sec
      Start 16: wrapping_integers_cmp
15/29 Test #16: wrapping_integers_cmp ............   Passed    0.01 sec
      Start 17: wrapping_integers_wrap
16/29 Test #17: wrapping_integers_wrap ...........   Passed    0.01 sec
      Start 18: wrapping_integers_unwrap
17/29 Test #18: wrapping_integers_unwrap .........   Passed    0.01 sec
      Start 19: wrapping_integers_roundtrip
18/29 Test #19: wrapping_integers_roundtrip ......   Passed    1.91 sec
      Start 20: wrapping_integers_extra
19/29 Test #20: wrapping_integers_extra ..........   Passed    0.50 sec
      Start 21: recv_connect
20/29 Test #21: recv_connect .....................   Passed    0.01 sec
      Start 22: recv_transmit
21/29 Test #22: recv_transmit ....................   Passed    0.80 sec
      Start 23: recv_window
22/29 Test #23: recv_window ......................   Passed    0.01 sec
      Start 24: recv_reorder
23/29 Test #24: recv_reorder .....................   Passed    0.02 sec
      Start 25: recv_reorder_more
24/29 Test #25: recv_reorder_more ................   Passed    1.38 sec
      Start 26: recv_close
25/29 Test #26: recv_close .......................   Passed    0.02 sec
      Start 27: recv_special
26/29 Test #27: recv_special .....................   Passed    0.02 sec
      Start 28: compile with optimization
27/29 Test #28: compile with optimization ........   Passed    1.94 sec
      Start 29: byte_stream_speed_test
             ByteStream throughput: 1.13 Gbit/s
28/29 Test #29: byte_stream_speed_test ...........   Passed    0.13 sec
      Start 30: reassembler_speed_test
             Reassembler throughput: 10.06 Gbit/s
29/29 Test #30: reassembler_speed_test ...........   Passed    0.14 sec
```

4. 提交代码

老样子，没什么好说的

5. 自己找点test case

交给测试做吧