浙江大学

本科实验报告

课程名称：　计算机网络基础

实验名称：　Lab5:the summit(TCP in full)

姓　　名：　蔡佳伟

学　　院：　计算机科学与技术学院

系：

专　　业：　软件工程

学　　号：　3220104519

指导教师：　高艺

2024 年 12 月 6 日

浙江大学实验报告

实验名称：Lab5:the summit(TCP in full)　实验类型：设计实验

同组学生：　　实验地点：

一、 实验背景：

- 学习掌握 TCP 的工作原理

二、 实验目的：

- 学习掌握 TCP 的工作原理
- 学习掌握 TCP connection 的相关知识
- 学习掌握协议栈结构

三、 实验内容

- 将 TCPSender 和 TCPReceiver 结合，实现一个 TCP 终端，同时收发数据。
- 实现 TCP connection 的状态管理，如连接和断开连接等。
- 整合网络接口、IP 路由以及 TCP 并实现端到端的通信。

四、 主要仪器设备

- 联网的 PC 机
- Linux 虚拟机

五、 操作方法与实验步骤

- 在开始整合代码之前，你需要了解以下 TCPConnection 的准则：

- 接收段，TCPConnection 的 **segment_received()**方法被调用时，它从网络中接收 TCPSegment。当这种情况发生时，TCPConnection 将查看段并且：

1. 如果设置了 RST 标志，将入站流和出站流都设置为错误状态，并永久终止连接，否则...

2. 把这个段交给 TCPReceiver，这样它就可以在传入的段上检查它关心的字段：seqno、SYN、负载以及 FIN。

3. 如果设置了 ACK 标志，则告诉 TCPSender 它关心的传入段的字段：ackno 和 window_size。

4. 如果传入的段占用了任何序列号，TCPConnection 确保至少有一个段作为应答被发送，以放映 ackno 和 window_size 的更新。

5.有一个特殊情况，你将不得不在 TCPConnection 的 **segment_received()**方法中处理：响应一个“keep-alive”段。对端可能会选择发送一个带有无效序列号的段，以查看你的 TCP 是否仍然有效，你的 TCPConnection 应该回复这些“keep-alive”段，即使它们不占用任何序列号。实现的代码如下：

```
if (_receiver.ackno().has_value() and (seg.length_in_sequence_space() == 0)
        and seg.header().seqno == _receiver.ackno().value() - 1) {
        _sender.send_empty_segment()
        }
```

- 发送段，TCPConnection 将通过网络发送 TCPSegment：
  1. TCPSender 将一个段推入它的传出队列，并设置了它在传出段上负责的字段（segno，SYN,负载以及 FIN）。
  2. 在发送段之前，TCPConnection 将向 TCPReceiver 询问它负责传出段的字段：ackno 和 window_size，如果有一个 ackno，它将设置 ACK 标志以及 TCPSegment 中的内容。

- 当时间流逝，TCPConnection 有一个 tick()方法，该方法将被操作系统定期调用，当这种情况发生时，TCPConnection 需要：
  1. 告诉 TCPSender 时间的流逝。
  2. 如果连续重传的次数超过上限 **TCPConfig::MAX_RETX_ATTEMPTS**，则终止连接，并发送一个重置段给对端（设置了 RST 标志的空段）。
  3. 如有必要，结束连接。

- TCPConnection 的一个重要功能是决定 TCP 连接何时结束。当这种情况发生时，实现释放它对本地端口号的独占声明，停止向传入的段发送应答，并让它的 active()方法返回 false。有两种方式可以结束连接，在不干净的关闭中，TCPConnection 发送或接收一个设置了 RST 标志的段。在这种情况下，出站和入站字节流都应该处于 error 状态，active()可以立即返回 false。干净的关闭是实现关闭而没有错误的方法，它尽可能地确保两个字节流的每一个都已完全可靠地传递给了接收的对端，与“远程”的对端的连接要完全关闭，有四个先决条件：
  1. 入站流已经完全组装完毕。
  2. 出站流已经被本地应用程序结束，并完全发送给对端。
  3. 出站流已被对端完全确认。
  4. 本地 TCPConnection 确信对端可以满足条件 3，这种情况发生后有两种选项：
     - 在两个流结束后逗留
     - 被动关闭

- TCPConnection 中有一个名为**_linger_after_streams_finish** 的成员变量，通过 state()方法向测试设备公开。变量开始时为 true。如果入站流在 TCPConnection 到达其出站流的 EOF 之前结束，则需要将此变量设置为 false。

- 在满足先决条件 1 到 3 的任何时刻，如果**_linger_after_streams_finish** 为 false，则连接“完成”（并且 active()应该返回 false）。否则，你需要等待：连接在足够的时间（10 * **_cfg.rt_timeout**）后完成。

- 你可能会觉得理解状态转换的过程有点困难，不用担心，我们已经为这部分提供了一些代码，你只需要确保你对它有一定的理解就可以很好地完成这个实验。

- 完成 TCPConnection 编写后，运行 **make check** 命令来自动测试整个实验所需实现代码的正确性。

- 重新测试 webget，修改 **webget.cc** 文件，把 TCPSocket 替换为 FullStackSocket，同时将**#include “socket.hh”**替换为**#include “tcp_sponge_socket.hh”**，并在你的**get_URL()**方法的结尾加上 **socket.wait_until_closed()**，重新进行测试。如果你遇到了什么问题，尝试运行**./apps/webget cs144.keithw.org /hasher/xyzzy** 并跟踪其的发送和接收。

- 最终测试，开启两个终端，在一个终端上运行./apps/lab4 server cs144.keithw.org 3000 的命令充当服务器，另一个终端运行./apps/lab4 client cs144.keithw.org 3001 命令充当客户端，如果成功连接，请尝试向对端发送信息并看看接收情况。最后在两个终端分别运行 ctrl D 来结束连接。

- 温馨提示：当你在开发代码的时候，可能会遇到无法解决的问题，下面给出解决的办法。

- 运行 **cmake .. -DCMAKE_BUILD_TYPE=RelASan** 命令配置 build 目录，使编译器能够检测内存错误和未定义的行为并给你很好的诊断。

- 你还可以使用 valgrind 工具。

- 你也可以运行 **cmake .. -DCMAKE_BUILD_TYPE=Debug** 命令配置并使用 GNU 调试器（**gdb**）。

- 你可以运行 **make clean** 和 **cmake .. -DCMAKE_BUILD_TYPE=Release** 命令重置构建系统。

- 如果你不知道如何修复遇到的问你题，你可以运行 **rm -rf build** 命令删除 build 目录，创建一个新的 build 目录并重新运行 **cmake..**命令。

六、 实验数据记录和处理

- 实现 TCPConnection 的关键代码截图

Tcp_connection.hh:

```cpp
class TCPConnection {
  private:
    TCPConfig _cfg;
    TCPReceiver _receiver{_cfg.recv_capacity};
    TCPSender _sender{_cfg.send_capacity, _cfg.rt_timeout, _cfg.fixed_isn};

    //! outbound queue of segments that the TCPConnection wants sent
    std::queue<TCPSegment> _segments_out{};

    //! Should the TCPConnection stay active (and keep ACKing)
    //! for 10 * _cfg.rt_timeout milliseconds after both streams have ended,
    //! in case the remote TCPConnection doesn't know we've received its whole
    bool _linger_after_streams_finish{true};

    size_t _time_since_last_segment_received=0;

    bool _is_active{true};

    void _set_rst_state(const bool send_rst);

    void _add_to_send_ackno_and_window();
```

添加计时器，初始化为 0，添加两个 private 函数，具体在.cc 中实现

Tcp_connection.cc:

libsponge > tcp_connection.cc > tick(const size_t)

```cpp
#include "tcp_connection.hh"

#include <iostream>

// Dummy implementation of a TCP connection

// For Lab 4, please replace with a real implementation that passes the
// automated checks run by `make check`.

template <typename... Targs>
void DUMMY_CODE(Targs &&... /* unused */) {}

using namespace std;

size_t TCPConnection::remaining_outbound_capacity() const {
    return _sender.stream_in().remaining_capacity();
}

size_t TCPConnection::bytes_in_flight() const {
    return _sender.bytes_in_flight();
}

size_t TCPConnection::unassembled_bytes() const {
    return _receiver.unassembled_bytes();
}

size_t TCPConnection::time_since_last_segment_received() const {
    return _time_since_last_segment_received;
}

void TCPConnection::segment_received(const TCPSegment &seg) {

    const auto& header=seg.header();
    _time_since_last_segment_received=0;

    // 1. 如果设置了RST标志，将入站流和出站流都设置为错误状态，并永久终止连接，否则...
    if(header.rst){
        _set_rst_state(false);
        return;
```

```
    }

    // 2. 把这个段交给TCPReceiver，这样它就可以在传入的段上检查它关心的字段：seqno、SYN、负载以及FIN
    _receiver.segment_received(seg);

    // 3. 如果设置了ACK标志，则告诉TCPSender它关心的传入段的字段：ackno和window_size
    bool need_send_ack=seg.length_in_sequence_space()>0?1:0;

    // 如果传入的段占用了任何序列号，
    // TCPConnection确保至少有一个段作为应答被发送，以放映ackno和window_size的更新。
    if(header.ack){
        _sender.ack_received(header.ackno,header.win);
        if(need_send_ack&&!_segments_out.empty()){
            need_send_ack=false;
        }
    }

    // // you code here.
    // //你需要考虑到ACK包、RST包等多种情况

    // //状态变化(按照个人的情况可进行修改)
    // // 如果是 LISEN 到了 SYN
    if (TCPState::state_summary(_receiver) == TCPReceiverStateSummary::SYN_RECV &&
        TCPState::state_summary(_sender) == TCPSenderStateSummary::CLOSED) {
        // 此时肯定是第一次调用 fill_window，因此会发送 SYN + ACK
        connect();
        return;
    }

    // // 判断 TCP 断开连接时是否时需要等待
    // // CLOSE_WAIT
    if (TCPState::state_summary(_receiver) == TCPReceiverStateSummary::FIN_RECV &&
        TCPState::state_summary(_sender) == TCPSenderStateSummary::SYN_ACKED)
        _linger_after_streams_finish = false;

    // // 如果到了准备断开连接的时候。服务器端先断
    // // CLOSED
```

```
    if (TCPState::state_summary(_receiver) == TCPReceiverStateSummary::FIN_RECV &&
        TCPState::state_summary(_sender) == TCPSenderStateSummary::FIN_ACKED && !_linger_after_streams_finish) {
        _is_active = false;
        return;
    }

    // 有一个特殊情况，你将不得不在TCPConnection的segment_received()方法中处理：
    // 响应一个“keep-alive”段。
    // 对端可能会选择发送一个带有无效序列号的段，以查看你的TCP是否仍然有效，
    // 你的TCPConnection应该回复这些“keep-alive”段，即使它们不占用任何序列号。
    if(_receiver.ackno().has_value()&&(seg.length_in_sequence_space()==0)
        &&seg.header().seqno==_receiver.ackno().value()-1){
            _sender.send_empty_segment();
            need_send_ack=false;
    }

    // 如果收到的数据包里没有任何数据，则这个数据包可能只是为了 keep-alive
    if (need_send_ack)
        _sender.send_empty_segment();

    _add_to_send_ackno_and_window();

}

bool TCPConnection::active() const { return _is_active; }

size_t TCPConnection::write(const string &data) {
    // 写入数据到发送缓冲区
    auto ret=_sender.stream_in().write(data);
    _sender.fill_window();
    _add_to_send_ackno_and_window();
    return ret;
}

//! \param[in] ms_since_last_tick number of milliseconds since the last call to this method
void TCPConnection::tick(const size_t ms_since_last_tick) {
    // 当时间流逝，TCPConnection有一个tick()方法，该方法将被操作系统定期调用
```

```cpp
    // 1. 告诉TCPSender时间的流逝
    _time_since_last_segment_received+=ms_since_last_tick;
    _sender.tick(ms_since_last_tick);

    // 2. 如果连续重传的次数超过上限TCPConfig::MAX_RETX_ATTEMPTS,
    // 则终止连接，并发送一个重置段给对端（设置了RST标志的空段）。
    if(_sender.consecutive_retransmissions()>TCPConfig::MAX_RETX_ATTEMPTS){
        while(!_sender.segments_out().empty()){
            _sender.segments_out().pop();
        }
        _set_rst_state(true);
        return;
    }

    _add_to_send_ackno_and_window();

    if(_linger_after_streams_finish&&
    TCPState::state_summary(_receiver)==TCPReceiverStateSummary::FIN_RECV&&
    TCPState::state_summary(_sender)==TCPSenderStateSummary::FIN_ACKED&&
    _time_since_last_segment_received>=10*_cfg.rt_timeout){
        _is_active=false;
        _linger_after_streams_finish=false;
    }
}

void TCPConnection::end_input_stream() {
    _sender.stream_in().end_input();
    // 输入流结束后可能需要发送 FIN 包
    _sender.fill_window();
    _add_to_send_ackno_and_window();
}

void TCPConnection::connect() {
    // 建立连接，发送SYN包
    _sender.fill_window();
    _add_to_send_ackno_and_window();
}
```

```cpp
TCPConnection::~TCPConnection() {
    try {
        if (active()) {
            cerr << "Warning: Unclean shutdown of TCPConnection\n";

            // Your code here: need to send a RST segment to the peer
            _set_rst_state(true);
        }
    } catch (const exception &e) {
        std::cerr << "Exception destructing TCP FSM: " << e.what() << std::endl;
    }
}

// 设置RST状态，发送RST包并关闭连接
void TCPConnection::_set_rst_state(const bool send_rst){
    if(send_rst){
        TCPSegment seg;
        seg.header().rst=true;
        seg.header().seqno=_sender.next_seqno();
        // 直接将包添加到输出队列
        _segments_out.emplace(std::move(seg));
    }
    _is_active=false;
    _receiver.stream_out().set_error();
    _sender.stream_in().set_error();
    _linger_after_streams_finish=false;
}

// 添加 ACK 序列号和窗口大小信息到待发送包中
void TCPConnection::_add_to_send_ackno_and_window(){
    while(!_sender.segments_out().empty()){
        // 遍历待发送的包，将 ACK 信息和窗口大小添加到包头
        auto seg=std::move(_sender.segments_out().front());
        _sender.segments_out().pop();
        if(_receiver.ackno().has_value()){
            seg.header().ack=true;
            seg.header().ackno=_receiver.ackno().value();
```

```
            seg.header().ackno=_receiver.ackno().value();
        }
        seg.header().win=min(static_cast<size_t>(numeric_limits<uint16_t>::max()),_receiver.window_size());
        _segments_out.emplace(std::move(seg));
    }
}
```

- 运行 make check 命令的运行结果截图

```
        Start 166: t_isnR_128K_8K_lL
161/164 Test #166: t_isnR_128K_8K_lL ..............   Passed
  0.78 sec
        Start 167: t_isnD_128K_8K_l
162/164 Test #167: t_isnD_128K_8K_l ...............   Passed
  1.03 sec
        Start 168: t_isnD_128K_8K_L
163/164 Test #168: t_isnD_128K_8K_L ...............   Passed
  0.34 sec
        Start 169: t_isnD_128K_8K_lL
164/164 Test #169: t_isnD_128K_8K_lL ..............   Passed
  1.51 sec

100% tests passed, 0 tests failed out of 164

Total Test time (real) =  49.99 sec
[100%] Built target check
cjwjiwang@cjwjiwang-virtual-machine:~/lab2/sponge/build$
```

s

- 重新编写 webget.cc 的代码截图

apps > webget.cc > get_URL(const string &, const string &)

```
#include "tcp_sponge_socket.hh"
#include "util.hh"

#include <cstdlib>
#include <iostream>

using namespace std;

void get_URL(const string &host, const string &path) {
    // Your code here.

    // You will need to connect to the "http" service on
    // the computer whose name is in the "host" string,
    // then request the URL path given in the "path" string.

    // Then you'll need to print out everything the server sends back,
    // (not just one call to read() -- everything) until you reach
    // the "eof" (end of file).

    FullStackSocket sock;
    sock.connect(Address(host, "http"));
    sock.write("GET " + path + " HTTP/1.1\r\n");
    sock.write("Host: " + host + "\r\n");
    sock.write("Connection: close\r\n");
    sock.write("\r\n");
    // sock.shutdown(SHUT_WR);

    while (!sock.eof()) {
        cout << sock.read();
    }
    // sock.close();

}
```

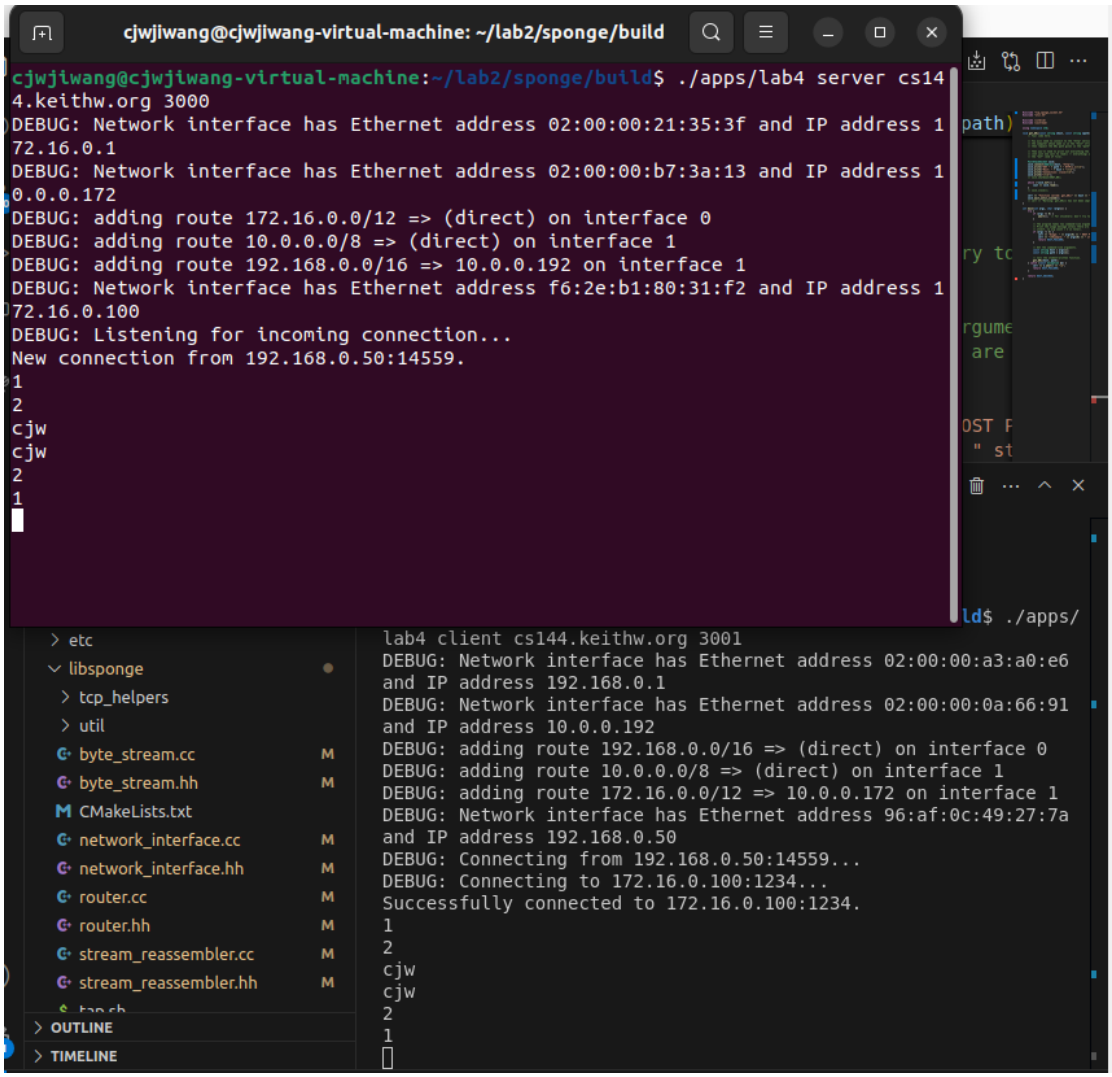
```
    cerr << "Function called: get_URL(" << host << ", " << path << ").\n";
    sock.wait_until_closed();
    // cerr << "Warning: get_URL() has not been implemented yet.\n";
}

int main(int argc, char *argv[]) {
    try {
        if (argc <= 0) {
            abort();  // For sticklers: don't try to access argv[0] if argc <= 0.
        }

        // The program takes two command-line arguments: the hostname and "path" part of the URL.
        // Print the usage message unless there are these two arguments (plus the program name
        // itself, so arg count = 3 in total).
        if (argc != 3) {
            cerr << "Usage: " << argv[0] << " HOST PATH\n";
            cerr << "\tExample: " << argv[0] << " stanford.edu /class/cs144\n";
            return EXIT_FAILURE;
        }

        // Get the command-line arguments.
        const string host = argv[1];
        const string path = argv[2];

        // Call the student-written function.
        get_URL(host, path);
    } catch (const exception &e) {
        cerr << e.what() << "\n";
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

- 重新测试 webget 的测试结果展示

```
 162/164 Test #167: t_isnD_128K_8K_l ..................   Passed    0.59 sec
         Start 168: t_isnD_128K_8K_L
 163/164 Test #168: t_isnD_128K_8K_L ..................   Passed    0.34 sec
         Start 169: t_isnD_128K_8K_lL
 164/164 Test #169: t_isnD_128K_8K_lL .................   Passed    0.73 sec

 100% tests passed, 0 tests failed out of 164

 Total Test time (real) =  48.38 sec
 [100%] Built target check
cjwjiwang@cjwjiwang-virtual-machine:~/lab2/sponge/build$
```

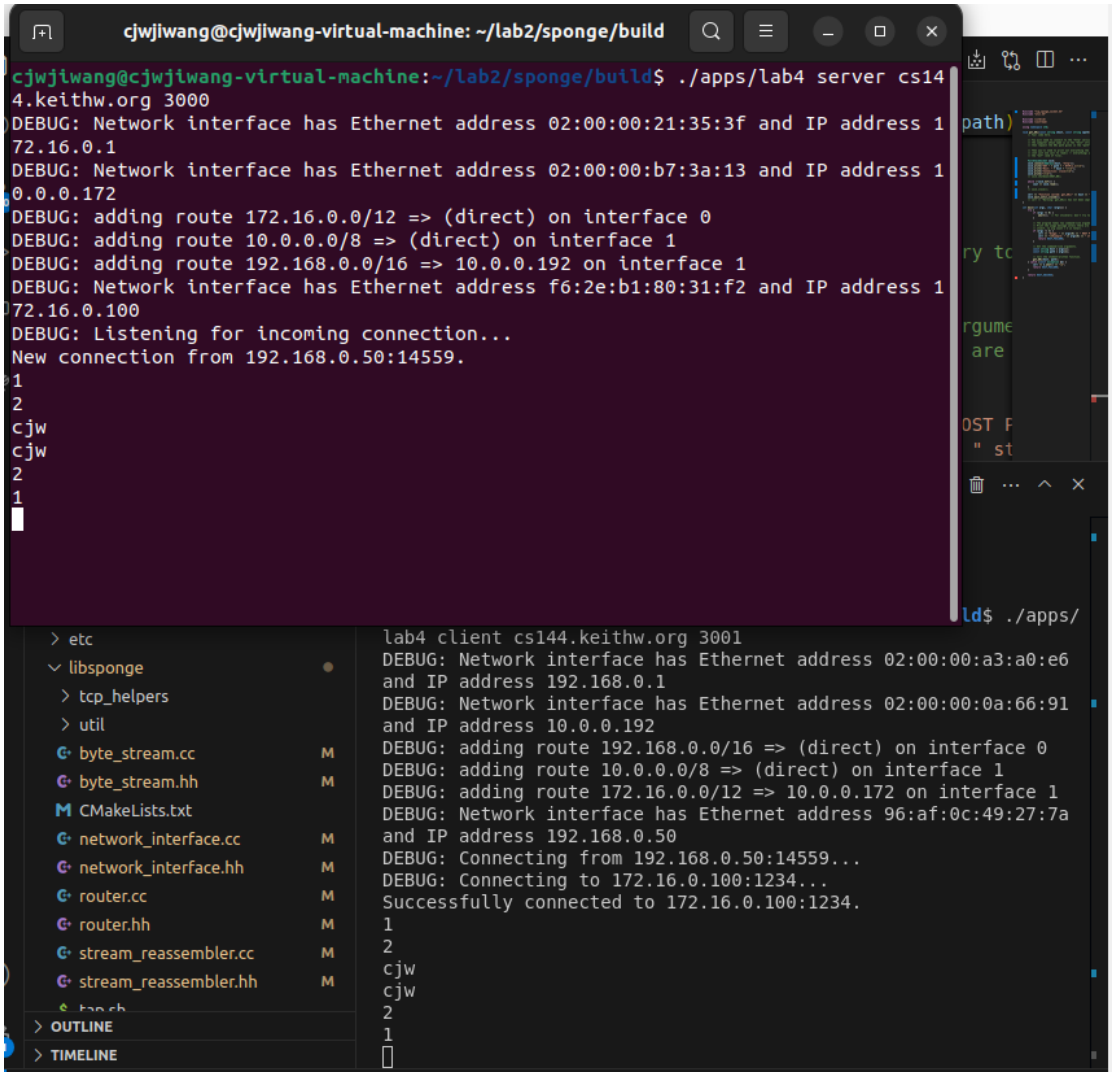
```
cjwjiwang@cjwjiwang-virtual-machine:~/lab2/sponge/build$ make check_webget
[100%] Testing webget...
Test project /home/cjwjiwang/lab2/sponge/build
    Start 31: t_webget
1/1 Test #31: t_webget ........................   Passed    1.09 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   1.09 sec
[100%] Built target check_webget
```

- 最终测试中服务器的运行截图



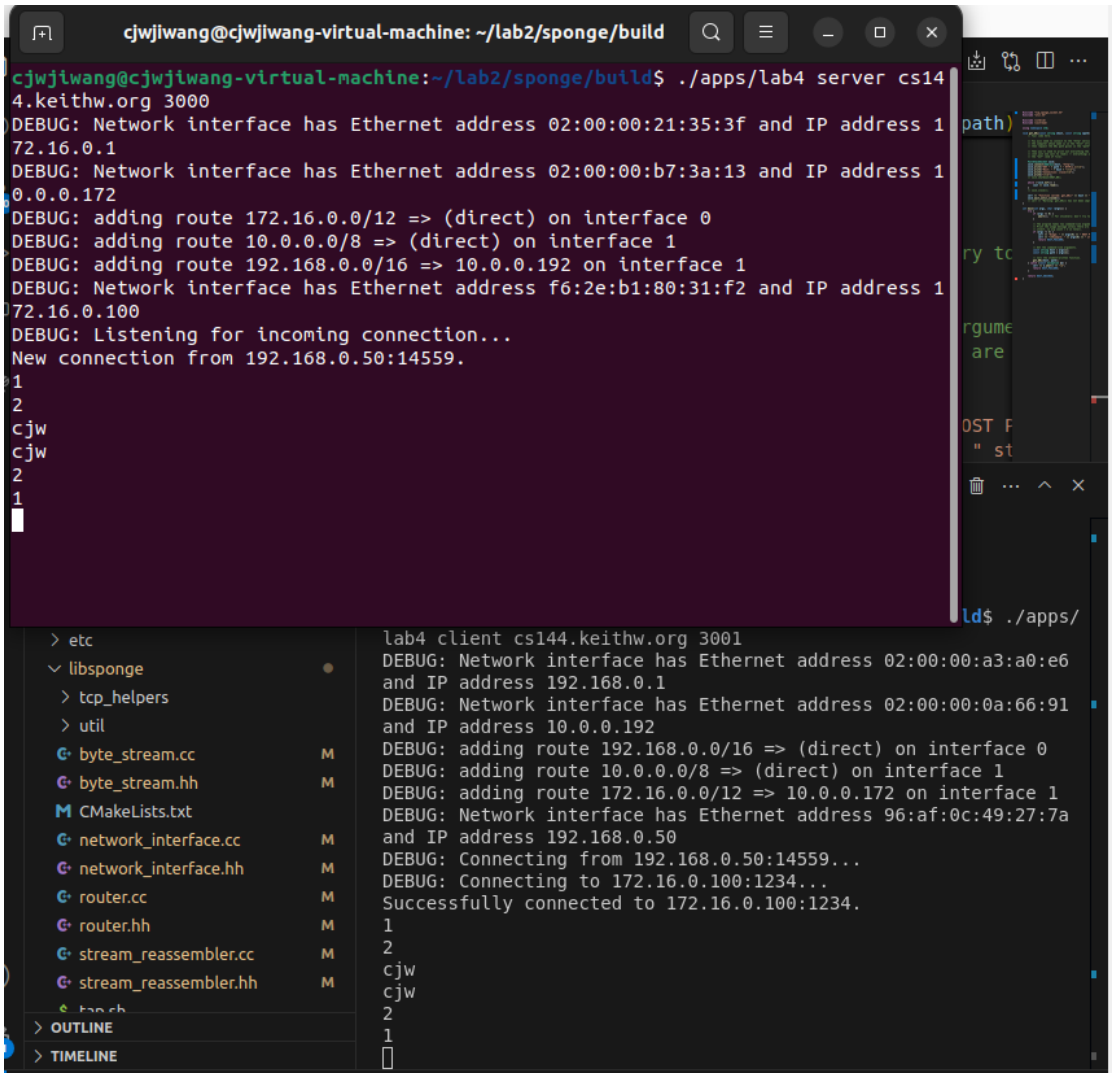
上面这个终端是服务器

- 最终测试中客户端的运行截图

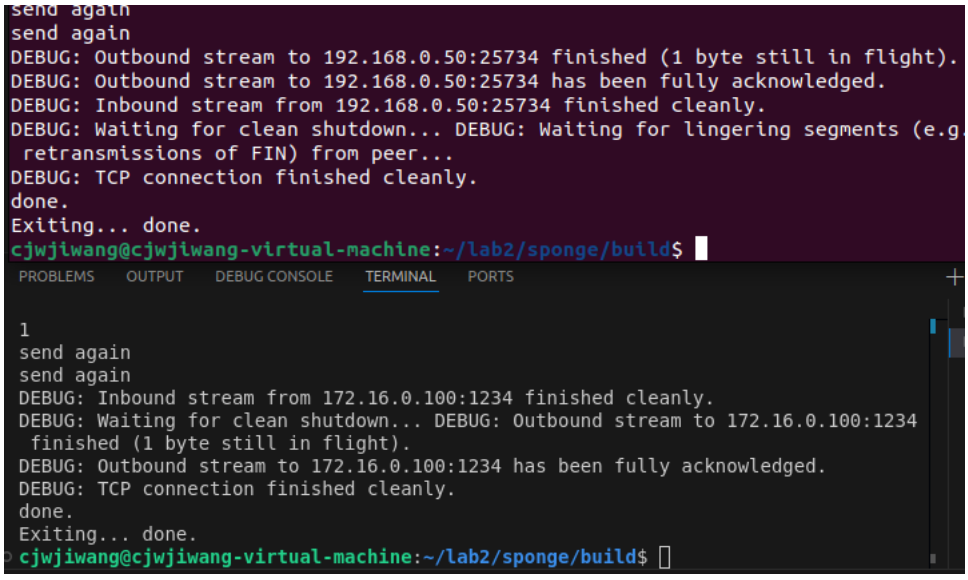


下面这个终端是客户端

- 在成功连接后端到端发送消息的运行截图


cjwjiwang@cjwjiwang-virtual-machine: ~/lab2/sponge/build
cjwjiwang@cjwjiwang-virtual-machine:~/lab2/sponge/build$ ./apps/lab4 server cs144.keithw.org 3000
DEBUG: Network interface has Ethernet address 02:00:00:21:35:3f and IP address 172.16.0.1
DEBUG: Network interface has Ethernet address 02:00:00:b7:3a:13 and IP address 10.0.0.172
DEBUG: adding route 172.16.0.0/12 => (direct) on interface 0
DEBUG: adding route 10.0.0.0/8 => (direct) on interface 1
DEBUG: adding route 192.168.0.0/16 => 10.0.0.192 on interface 1
DEBUG: Network interface has Ethernet address f6:2e:b1:80:31:f2 and IP address 172.16.0.100
DEBUG: Listening for incoming connection...
New connection from 192.168.0.50:14559.
1
2
cjw
cjw
2
1
etc
libsponge
tcp_helpers
util
byte_stream.cc
byte_stream.hh
CMakeLists.txt
network_interface.cc
network_interface.hh
router.cc
router.hh
stream_reassembler.cc
stream_reassembler.hh
OUTLINE
TIMELINE
lab4 client cs144.keithw.org 3001
DEBUG: Network interface has Ethernet address 02:00:00:a3:a0:e6 and IP address 192.168.0.1
DEBUG: Network interface has Ethernet address 02:00:00:0a:66:91 and IP address 10.0.0.192
DEBUG: adding route 192.168.0.0/16 => (direct) on interface 0
DEBUG: adding route 10.0.0.0/8 => (direct) on interface 1
DEBUG: adding route 172.16.0.0/12 => 10.0.0.172 on interface 1
DEBUG: Network interface has Ethernet address 96:af:0c:49:27:7a and IP address 192.168.0.50
DEBUG: Connecting from 192.168.0.50:14559...
DEBUG: Connecting to 172.16.0.100:1234...
Successfully connected to 172.16.0.100:1234.
1
2
cjw
cjw
2
1

- 成功关闭连接的截图


send again
send again
DEBUG: Outbound stream to 192.168.0.50:25734 finished (1 byte still in flight).
DEBUG: Outbound stream to 192.168.0.50:25734 has been fully acknowledged.
DEBUG: Inbound stream from 192.168.0.50:25734 finished cleanly.
DEBUG: Waiting for clean shutdown... DEBUG: Waiting for lingering segments (e.g. retransmissions of FIN) from peer...
DEBUG: TCP connection finished cleanly.
done.
Exiting... done.
cjwjiwang@cjwjiwang-virtual-machine:~/lab2/sponge/build$
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
1
send again
send again
DEBUG: Inbound stream from 172.16.0.100:1234 finished cleanly.
DEBUG: Waiting for clean shutdown... DEBUG: Outbound stream to 172.16.0.100:1234 finished (1 byte still in flight).
DEBUG: Outbound stream to 172.16.0.100:1234 has been fully acknowledged.
DEBUG: TCP connection finished cleanly.
done.
Exiting... done.
cjwjiwang@cjwjiwang-virtual-machine:~/lab2/sponge/build$

七、 实验数据记录和处理

- ACK 标志的目的是什么？ackno 是经常存在的吗？

  在 TCP 协议中，ACK（Acknowledgment）标志表示接收端已经成功接收了发送端发送的数据。发送端每发送一个数据包，接收端都需要发送一个带 ACK 标志的包作为确认。这样可以确保数据在网络中传输的可靠性，避免数据丢失和重传的问题。

  ackno 通常是经常存在的。每当接收端收到数据包并需要进行确认时，它都会在 TCP 段头中设置 ackno 字段。特别是在以下几种情况下：

  （1）当接收端收到包含有效数据的 TCP 段时，它会发送带 ackno 的确认包

  （2）当接收端仅接收到空数据（Keep-Alive 包）时，也可能发送空的 ACK 包

- 请描述 TCPConnection 的代码中是如何整合 TCPReceiver 和 TCPSender 的。

  在 segment_received()方法中：

  当接收端收到一个 TCP 段时，数据会交给_receiver 对象进行处理。

  如果收到带 ACK 标志的数据包，代码会调用_send.ack_received()方法来更新发送端的 ACK 状态，然后通过_add_to_send_ackno_and_window()方法将新的 ACK 序列号和窗口大小信息添加到待发送的 TCP 段中。

  在 write()方法中：

  先把数据写入发送流，然后调用_sender.fill_window()方法填充发送窗口。再通过_add_to_send_ackno_and_window()方法将窗口大小和 ACK 序列号添加到发送数据包头部。这样保证了发送端和接收端之间的数据流量控制、窗口调整等信息保持同步。

  TCP 状态转换有 state_summary()方法进行状态转换的检查。当接收端状态和发送端状态达到一定条件时，会进行状态转换，例如进入 CLOSE WAIT 状态。

- 在最终测试中服务器和客户端能互连吗？如果不能，你分析是什么原因？

  可以互连。如果不能，可能是网络原因或代码错误，比如状态转换逻辑出现错误等。

- 在关闭连接的时候是否两端都能正常关闭？如果不能，你分析是原因？

  可以正常关闭，如果不能，可能是因为 RST 包发送错误，强制关闭失败，或者主动与被动关闭不匹配等。

八、 实验数据记录和处理

我认为这段代码完成的文字提示非常明显地展示了面向状态机的设计理念，尤其是在 segment_received()和 tick()函数中体现得尤为突出。连接的每个状态（如 SYN_RECV、FIN_RECV、CLOSED 等）都通过 TCPState::state_summary()函数进行检查，并根据当前状态做出相应的动作。这种设计可以清晰地控制连接的生命周期，确保各个状态之间的转换是安全且符合协议的。
在 segment_received()函数中，空 ACK 包的发送逻辑让我印象深刻。对于那些没有实际数据的包（如 Keep-Alive 包），系统依然会发送空 ACK 包，保持连接的活跃性。如果不发送空包，测试点会过不去。
这种设计非常贴合真实的 TCP 协议，保证了在不传输数据的情况下，连接仍然能保持活跃并避免因长时间无通信而被认为是死连接。
在上次实验中，我的 sender 和 reveiver 文件是实现得比较好的，所以没有出特别大的问题。