

Computer Networks Lab0

黄嘉祺 221220108

1 Program Structure and Design

1.1 Reassembler

- 我在阅读了 reassembler.cc , reassembler.hh 及 \tests 中的源码后, 结合实验手册和ppt, 确定了 reassembler 类型的实验思路和实现方法
- 首先我在 reassembler.hh 中的了 reassembler 类里补充定义了 `std::map<uint64_t, std::string> buf_` 用来作为 reassembler 的缓存, 并定义了 `uint64_t tail_index` 用于指示最后一个substring是否已被缓存/推入
- 对于 reassembler.cc 中的 insert 函数, 我的思路如下:
 - 每次调用时, 对于接收到的substring data 及其起始下标 first_index , 定义 `last_index = first_index + data.length()` , 首先检测 last_index 是否小于 first_unassembled_index , 若小于则意味着该 data 已经被推入 bytestream , 故可以舍弃, 直接返回
 - 接下来检测该 first_index 是否小于 first_unassembled_index , 若小于则意味着该 data 的起始位置在 bytestream 的前面, 故将不与 bytestream 中重合的部分且未超出 first_unacceptable_index 的部分存入缓存 buf_ 中, 其对应起始下标为 first_unassembled_index , 注意在缓存时若对应起始下标已有数据块, 则用更大的数据块更新值, 若对应下标无数据块, 则创建新的键值对存储该数据块
 - 若 first_index 大于等于 first_unassembled_index , 则将 data 存入缓存 buf_ , 其对应起始下标为 first_index , 缓存时的更新规则同上
 - 若 first_index 大于等于 first_unacceptable_index , 则可以舍弃该数据块, 直接返回
 - 下图为 insert 第一部分: 更新缓存

```

void Reassembler::insert( uint64_t first_index, string data, bool is_last_substring )
{
    // put the substring into the buffer
    uint64_t last_index = first_index + data.size();
    if ( is_last_substring ) {
        tail_index = last_index;
        if ( tail_index == first_unassembled_index ) {
            output_.writer().close();
            return;
        }
    }
    if ( last_index <= first_unassembled_index ) {
        return;
    }
    else if ( first_index < first_unassembled_index ) {
        std::string data_to_buffer
            = data.substr( first_unassembled_index - first_index, first_unacceptable_index - first_unassembled_index );
        if ( buf_.find( first_unassembled_index ) == buf_.end() ) {
            buf_[first_unassembled_index] = data_to_buffer;
        }
        else {
            if ( data_to_buffer.size() > buf_[first_unassembled_index].size() ) {
                buf_[first_unassembled_index] = data_to_buffer;
            }
        }
    }
    else if ( first_index >= first_unassembled_index && first_index < first_unacceptable_index ) {
        std::string data_to_buffer = data.substr( 0, first_unacceptable_index - first_index );
        if ( buf_.find( first_index ) == buf_.end() ) {
            buf_[first_index] = data_to_buffer;
        }
        else {
            if ( data_to_buffer.size() > buf_[first_index].size() ) {
                buf_[first_index] = data_to_buffer;
            }
        }
    }
    else {
        return;
    }
}

```

- 接下来维护 buf_，将 buf_ 中所有可以合并（存在交叉、包含关系）的数据块均合并，保证下一次调用并更新 buf_ 前缓存中的数据块均不相交
- 下图为 insert 第二部分：合并相交数据块，维护缓存

```

// merge and maintain the buffer
if ( !buf_.empty() ) {
    auto it = buf_.begin();
    while ( it != buf_.end() ) {
        auto it2 = it;
        it2++;
        while ( it2 != buf_.end() ) {
            if ( it->first + it->second.size() >= it2->first ) {
                it->second = merge_strings( it->second, it->first, it2->second, it2->first );
                it2 = buf_.erase( it2 );
            }
            else {
                it2++;
            }
        }
        it++;
    }
}
}

```

- 最后检测 buf_ 中是否存在可以推入 byte_stream（起始下标等于 first_unassembled）的数据块，则将该数据块推入 byte_stream，将其从 buf_ 中删除
- 关闭 output_.writer 条件：**1)** 在函数一开始检查 is_last_string 是否为真，若为真意味着最后一个substring已被插入，则将 tail_index 赋值为 last_index，并检测若此时 tail_index 等于 first_unassembled_index 则关闭 output_.writer **2)** 在函数的最后检测若 tail_index 等于 first_unassembled_index 且 buf_ 为空，则关闭 output_.writer
- 下图为 insert 第三部分：推入 byte_stream

```

// see whether we can write to the output
if ( !buf_.empty() ) {
    auto first_buf = buf_.begin();
    if ( first_buf->first == first_unassembled_index ) {
        output_.writer().push( first_buf->second );
        buf_.erase( first_buf );
    }
}
if ( buf_.empty() && tail_index == first_unassembled_index ) {
    output_.writer().close();
}
// Your code here.
// (void)first_index;
// (void)data;
// (void)is_last_substring;
}

```

- 对于 reassembler.cc 中的 bytes_pending 函数，思路则很简单：
 - 该函数的功能是返回当前缓存中的总数据字节数，即 buf_ 中所有数据块的长度之和
 - 遍历 buf_ 中的所有数据块，将其长度累加并返回
 - 下图为 bytes_pending 函数

```

uint64_t Reassembler::bytes_pending() const
{
    // Your code here.
    // return {};
    uint64_t res = 0;
    for ( auto s : buf_ ) {
        res += s.second.size();
    }
    return res;
}

```

2 Implementation Challenges

- 在实现 buf_ 的时候，一开始我使用的是 std::vector<std::string> 类型数据结构来存储数据块，并用 std::vector<uint64_t> 数据结构来存储每个数据块对应起始下标，然而这样由于不方便直接查询修改和删除，并且无法自动将数据块按照起始下标排序，故导致时间代价较高，且由于写法十分复杂（代码近100行），导致不仅性能差，还出现了不少错误
- 我将 buf_ 改为用 std::map<uint64_t, std::string> 来存储数据块，这样可以直接通过起始下标查询修改和删除，且 std::map 会自动将数据块按照起始下标排序，这样代码简洁明了，性能也大幅度提升

3 Remaining Bugs

目前我的代码已经通过了全部的测试，暂时未发现明显的bug

4 Experimental Results and Performance

- Bytestream and reassembler: Passed all 17 tests

```
3/17 Test #4: byte_stream_capacity ..... Passed 0.02 sec
      Start 5: byte_stream_one_write
4/17 Test #5: byte_stream_one_write ..... Passed 0.02 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 0.06 sec
      Start 8: byte_stream_stress_test
7/17 Test #8: byte_stream_stress_test ..... Passed 0.03 sec
      Start 9: reassembler_single
8/17 Test #9: reassembler_single ..... Passed 0.02 sec
      Start 10: reassembler_cap
9/17 Test #10: reassembler_cap ..... Passed 0.02 sec
      Start 11: reassembler_seq
10/17 Test #11: reassembler_seq ..... Passed 0.03 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 0.02 sec
      Start 14: reassembler_overlapping
13/17 Test #14: reassembler_overlapping ..... Passed 0.02 sec
      Start 15: reassembler_win
14/17 Test #15: reassembler_win ..... Passed 0.53 sec
      Start 37: compile with optimization
15/17 Test #37: compile with optimization ..... Passed 0.08 sec
      Start 38: byte_stream_speed_test
      ByteStream throughput: 2.59 Gbit/s
16/17 Test #38: byte_stream_speed_test ..... Passed 0.11 sec
      Start 39: reassembler_speed_test
      Reassembler throughput: 5.84 Gbit/s
17/17 Test #39: reassembler_speed_test ..... Passed 0.17 sec

100% tests passed, 0 tests failed out of 17

Total Test time (real) = 1.54 sec
Built target check1
○ Lefty777@EDVAC-2023:~/minnow$
```