计算机网络 Project - Design

18302010018 俞哲轩

项目结构

> 新建了以下文件

文件	用途
packet.h	定义一个数据包的结构，包括头部以及数据部分 根据文档定义一些常量，包括头部的***Magic Number***、包各部分的***Length***以及数据包的***Type***和与之对应的***Code***
list.[h|c]	定义一个单向链表的数据结构 用于实现发送方的***IHAVE<list>***和接收方的***WHOHAS<list>***
init. [h|c]	定义了chunk的结构以及初始化函数 读取每一个peer的***master-chunk-file***和***has-chunk-file***的信息
conn. [h|c]	发送方和接收方创建可靠传输连接的函数 包括发送/接收池***snd_conn***、***rcv_conn***以及发送/接收池***snd_pool***、***rcv_pool***
util. [h|c]	事务处理函数 处理包括***GET chunk***、***snd_packet***、***rcv_ACK***以及***timeout***

具体设计

可靠传输 (Reliable Transfer)

发送方和接收方采取的基本策略如下：

1. 发送方将**超时（timeout）**和**收到3个冗余ACK（3 duplicate ACKs）**均定义为**丢包**事件，将触发**重传**机制
2. 接收方使用**GBN**策略，**无需维护接收窗口**且**累计确认**

并发传输 (Concurrent Transfer)

Bit-Torrent的并发传输和一般的并发线程（Thread）**不同**，并非是指同时执行多个线程，而是**同时从多个peer这里进行下载**；在实现中，采用了类似于线程池的思想：

1. 接收方维护一个**rcv_pool**，表明从**哪一个peer**执行下载，获取**哪一部分chunk**
2. 发送方维护一个**snd_pool**，表明对**哪一个peer**执行发送，传输**哪一部分chunk**

鲁棒性 (Robustness)

对于可能发生的意外事情，设计了如下处理：

Peer crashes

为了应对peer crashes的情况，设计了如下处理办法：

1. 每次收到3个冗余ACK或者超时，都会触发重传机制
2. 区别对待这两个触发原因：收到3个冗余ACK代表**丢包**（即连接还是**可用**的）、超时代表**拥塞**或者连接可能**不可用**（即**peer crashes**）
3. 维护两个变量**duplicate_ACK_cnt**和**timeout_cnt**，每次触发重传的时候，相应的重传原因**数值加1**
4. 当**多次重传没有反应**（即记录重传原因是超时的变量**数值超过5**的时候），认定为**peer crashed**
5. 认定为**peer crashed**之后，**终止当前下载**，**重新发送WHOHAS**，重新下载

General robustness

实现基本的鲁棒性，确保每次下载的内容不会出错，设计了如下校验方法：

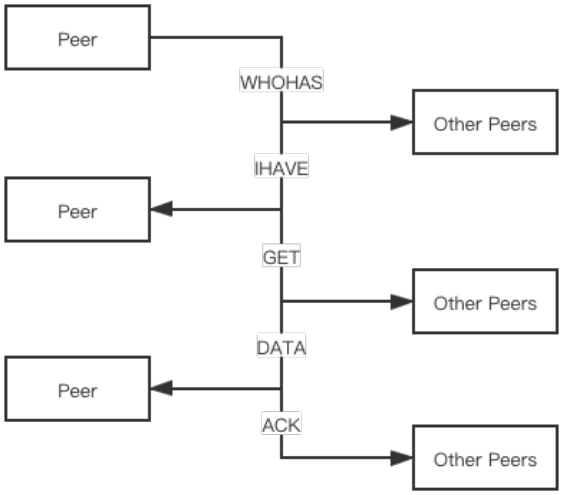
1. 每次接收方下载完成一个chunk，就通过sha**比对哈希校验**
2. 如果所有的哈希校验都**比对成功**，则代表下载的文件没有错误
3. 如果有一个chunk**比对失败**，**终止当前下载**，**重新发送WHOHAS**，重新下载

程序逻辑

当一个peer接收到来自用户命令行的 GET 任务请求时，会执行以下步骤：

1. 当前peer会向其他所有peers发送**WHOHAS分组**
2. 其他peers接收到WHOHAS分组，检查自己已有的分组，返回**IHAVE分组**
3. 当前peer接收到IHAVE分组，创建连接并向对应的peer发送**GET分组**
4. 其他peers接收到GET分组，发送**DATA分组**，并启动timer，如果丢包则重传
5. 当前peer完整接收到所有DATA分组之后，比对校验，如果正确，则传输完成

具体的**程序流程图**如下：



代码实现

packet.h

根据文档，定义了包头部和包种类

Packet Type	Code
WHOHAS	0
IHAVE	1
GET	2
DATA	3
ACK	4
DENIED	5

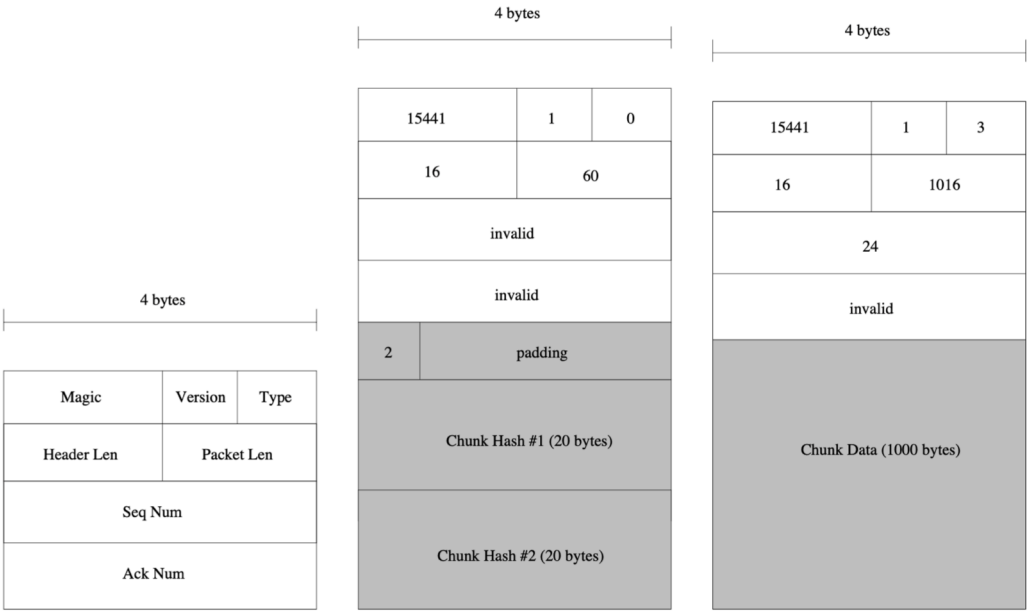
```
// Package Header
#define HEADER_LEN          16
#define PACKET_LEN          1500
#define DATA_LEN            (PACKET_LEN - HEADER_LEN)
#define MAGIC_NUM           15441
#define MAX_CHUNK_NUM       74

// Packet Type & Code
#define WHOHAS              0
#define IHAVE               1
#define GET                 2
#define DATA                3
#define ACK                 4
#define DENIED              5

// Define Header
typedef struct header_s {
  unsigned short magic;
```

```
  unsigned char version;
  unsigned char type;
  unsigned short header_len;
  unsigned short packet_len;
  unsigned int seq_num;
  unsigned int ack_num;
} header_t;

// Define Packet
typedef struct packet_s {
  header_t header;
  char data[DATA_LEN];
} packet_t;
```



(a) The basic packet header, with each header field named.

(b) A full WHOHAS request with two Chunk hashes in the request. Note that both seq num and ack num have no meaning in this packet.

(c) A full DATA packet, with seq number 24 and 1000 bytes of data. Note that the ack num has no meaning because data-flow is one-way.

list.[h|c]

定义了一个单向链表的数据结构

```
typedef struct node_s {
    void *data;
    struct node_s *next;
} node_t;

typedef struct list_s {
    int node_num;
    node_t *head;
    node_t *tail;
} list_t;
```

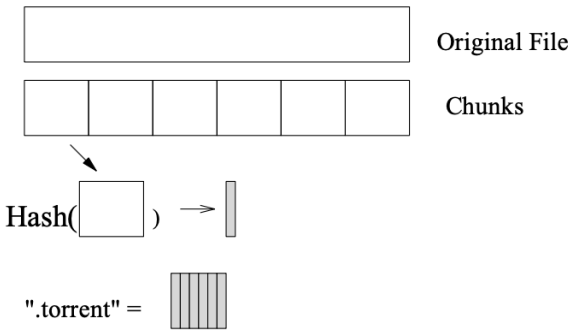
```
list_t *init_list(); // init list

void add_node(list_t *list, void *data); // add node

void *remove_node(list_t *list); // remove node
```

init.[h|c]

根据文档，定义了每一个chunk的结构，包括id和chunk内容的hash，得到如下结构（一个文件的所有chunk的hash拼接起来，成为torrent文件）：




```
typedef struct chunk_s {
    int id;
    char chunk_hash[SHA1_HASH_SIZE];
} chunk_t;

void init_tracker(); // read master-chunk-file

void init_chunks_ihave(); // read has-chunk-file
```

conn.[h|c]

发送方连接包括：

- 窗口大小（恒等于8）
- 上一个接收到的ACK编号
- 下一个准备发送的分组的序号
- 在发送窗口后一个包的序号
- 冗余ACK的计数器

发送方连接池则包括了该peer所有的发送方连接

```
typedef struct snd_conn_s {
  int cwnd; // window size
  int last_ack; // last ack received
  int next_to_send; // next to send
  int available; // index of next packet after the send window
  int dup_times; // duplicate times of last_ack
```

```
  long begin_time;
  packet_t **pkts; // cache all packets to send
  bt_peer_t *receiver;
} snd_conn_t;

typedef struct snd_pool_s {
  snd_conn_t **conns;
  int conn_num;
  int max_conn;
} snd_pool_t;
```

接收方连接包括：

- 缓存数据的chunk buffer
- 记录写入buffer位置的编号
- 下一个等待发送的ACK编号

接收方连接池则包括了该peer所有的接收方连接

```
typedef struct rcv_conn_s {
  chunk_buffer_t *chunk_buf; // use to cache data
  int from_here; // use when caching data into chunk_buf
  int next_ack; // next ack expected
  bt_peer_t *sender;
} rcv_conn_t;

typedef struct rcv_pool_s {
  rcv_conn_t **conns;
  int conn_num;
  int max_conn;
} rcv_pool_t;
```

建立send和receive连接的方法基本相同，基本步骤即为：

- 新建发送/接收连接
- 加入发送/接收池
- 关闭连接并移出发送/接收池

以receive为例：

```
rcv_conn_t *init_rcv_conn(bt_peer_t *peer, chunk_buffer_t *chunk_buf) {
  rcv_conn_t *conn = malloc(sizeof(rcv_conn_t));
  conn->from_here = 0;
  conn->next_ack = 1;
  conn->chunk_buf = chunk_buf;
  conn->sender = peer;
  return conn;
}
```

```
snd_conn_t *init_snd_conn(bt_peer_t *peer, packet_t **pkts); // almost the same

rcv_conn_t *add_to_rcv_pool(rcv_pool_t *pool, bt_peer_t *peer, chunk_buffer_t
*chunk_buf) {
  rcv_conn_t *conn = init_rcv_conn(peer, chunk_buf);
  for (int i = 0; i < pool->max_conn; i++) {
    if (pool->conns[i] == NULL) {
      pool->conns[i] = conn;
      break;
    }
  }
  pool->conn_num++;
  return conn;
}

snd_conn_t *add_to_snd_pool(snd_pool_t *pool, bt_peer_t *peer, packet_t
**pkts); // almost the same

void remove_from_rcv_pool(rcv_pool_t *pool, bt_peer_t *peer) {
  rcv_conn_t **conns = pool->conns;
  for (int i = 0; i < pool->max_conn; i++) {
    if (conns[i] != NULL && conns[i]->sender->id == peer->id) {
      free_chunk_buffer(conns[i]->chunk_buf);
      free(conns[i]);
      conns[i] = NULL;
      pool->conn_num--;
      break;
    }
  }
}

void remove_from_snd_pool(snd_pool_t *pool, bt_peer_t *peer); // almost the
same
```

util.[h|c]

`manage_user_input`方法处理用户输入的**GET指令**，根据用户的“GET chunk-file target-file”指令，开始任务处理

`new_whohas_pkt`方法根据用户指令，生成一个WHOHAS分组

```
void manage_user_input(char *chunk_file, char *out_file); // handle user's GET
command

list_t *new_whohas_pkt(); // create WHOHAS packets according to the GET command
```

`add_sender`方法把发送下一个chunk的sender更新至peer，准备好下一个chunk sender

```
char *add_sender(list_t *chunk_hash_list, bt_peer_t *peer); // add next chunk sender
```

`check_and_add_data`校验哈希值检查下载的文件是否出错，如果没有出错，则完成传输

```
void check_and_add_data(char *hash, char *data); // check the chunk data by the hash
```

`manage_whohas`方法处理**WHOHAS分组**，寻找自己有没有这个chunk，如果有，发送一个IHAVE分组，否则返回NULL

```
packet_t *manage_whohas(packet_t *pkt_whohas); // return a IHAVE packet if this peer has the chunk
```

`chunk_hash_to_torrent`方法将hash组合成一个.torrent文件

`split_into_chunks`方法将.torrent文件分拆成chunks

```
void chunk_hash_to_torrent(char *payload, int chunk_num, chunk_t *chunks); // assemble hashes to torrent file

list_t *split_into_chunks(void *payload); // split torrent file into chunks
```

`manage_ihave`方法处理**IHAVE分组**，先调用 `add_sender`，把准备发送下一个chunk的sender更新至peer，再把要下载的chunk hash创建connection

```
packet_t *manage_ihave(packet_t *pkt, bt_peer_t *peer); // manage IHAVE
```

`void manage_get`方法处理**GET分组**，并发送相应的chunk data

```
void manage_get(int sock, packet_t *pkt, bt_peer_t *peer); // manage GET

void send_data_pkts(snd_conn_t *conn, int sock, struct sockaddr *to); // send data
```

`manage_data`方法处理**DATA分组**，并接收相应的chunk data

```
void manage_data(int sock, packet_t *pkt, bt_peer_t *peer); // manage DATA

packet_t **get_data_pkts(char *chunk_hash); // receive data
```

`manage_ack`方法处理ACK分组，如果是正常的ACK，则发送窗口后移；如果是冗余ACK，则进行累计，超过3次，则重发所有未被确认的数据包

```
void manage_ack(int sock, packet_t *pkt, bt_peer_t *peer); // manage ACK
```

`manage_timeout`方法处理超时的情况，发生超时，则重发所有未被确认的数据包

```
void manage_timeout(); // manage TIMEOUT
```

BONUS

自动测试

> /Starter Code/auto_test.sh

提供了自动测试脚本，在Starter Code目录下，输入`./auto_test.sh`即可自动测试所有checkpoints；如果遇到permission denied错误，执行`chmod 777 auto_test.sh`给予权限即可

拥塞控制

在测试checkpoint 3 test2的时候，发现了存在**大量queue loss**的情况，猜测可能是网络发生了拥塞，导致的**大量丢包**的出现

结合课程中学习到的**TCP拥塞控制**知识，实现了一个简单的**TCP Tahoe**

代码改变部分如下：

- 创建发送方连接的时候，设置窗口初始值为1
- 设置slow start threshold初始值为64

```
// in conn.c
snd_conn_t *init_snd_conn(bt_peer_t *peer, packet_t **pkts) {
    snd_conn_t *conn = malloc(sizeof(snd_conn_t));
    conn->last_ack = 0;
    conn->to_send = 0;
    conn->dup_times = 0;
    conn->cwnd = 1; // initial congestion control window size
    conn->ssthresh = 64; // initial slow start threshold
    conn->available = 1;
    conn->rtt_flag = 1;
    conn->begin_time = clock();
    conn->receiver = peer;
    conn->pkts = pkts;
    return conn;
}
```

- 在接收到ACK的时候，判断是否是冗余ACK，如果不是冗余ACK，则发送窗口先**翻倍**，超过ssthresh之后**线性增长**；如果是冗余ACK，累计个数，超过3次则发送窗口**减少为1**，ssthresh**减半**
- 在timeout的时候，则发送窗口**减少为1**，ssthresh**减半**

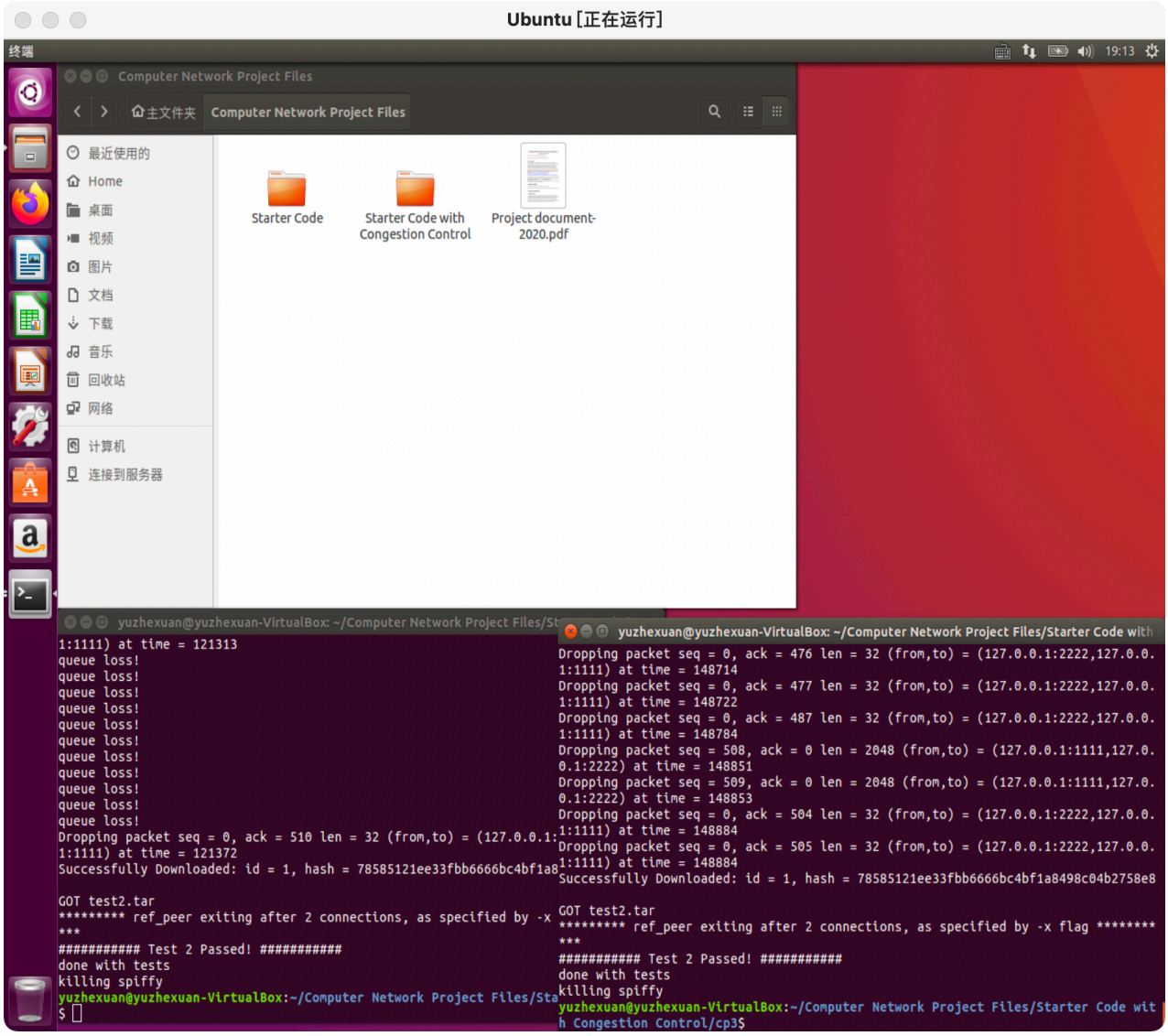
```
// in util.c manage_ACK()
if (snd_conn->cwnd < snd_conn->ssthresh) { // slow start
  snd_conn->cwnd += add_wnd;
  snd_conn->rtt_flag = ack_num + snd_conn->cwnd;
```

```
} else { // congestion avoidance
  if (ack_num >= snd_conn->rtt_flag) {
    snd_conn->cwnd += 1;
    snd_conn->rtt_flag += snd_conn->cwnd;
  }
}
snd_conn->to_send = snd_conn->to_send > ack_num ? snd_conn->to_send : ack_num;

if (snd_conn->dup_times >= 3) { // 3 duplicate ACKs
  int snd_ssthresh = (int) (snd_conn->cwnd / 2);
  snd_conn->ssthresh = snd_ssthresh >= 2 ? snd_ssthresh : 2;
  snd_conn->cwnd = 1;
}

// in util.c manage_timeout()
if (timeout) { // timeout
  int snd_ssthresh = (int) (this_snd_conn->cwnd / 2);
  this_snd_conn->ssthresh = snd_ssthresh >= 2 ? snd_ssthresh : 2;
  this_snd_conn->cwnd = 1;
}
```

两者对比（左边为没有拥塞控制，右边为实现拥塞控制）：



实现了拥塞控制的版本，几乎**没有出现queue loss**的情况，但是所用时间，比没有实现拥塞控制的版本要**慢**，可能原因是：拥塞控制将发送方窗口大小限制在**8以下**，发送速率较慢，故用时较长