计算机网络 Project 2

说明文档结构：

1. 代码结构
2. 数据结构
3. 程序逻辑
4. 关于超时与拥塞控制
5. 关于并发运行
6. 关于开发过程中的问题
7. 程序运行结果图

代码结构

1. peer.c:程序的主结构，主函数入口，实现了对用户输入命令的处理和以及对接收到的不同类型数据包处理
2. queue.c/h：队列数据结构的封装文件并实现了队列的一系列操作的函数
3. task.c/h：上传和下载任务的函数分装文件，定义了任务、连接等结构，实现了对上传和下载的控制操作函数
4. timer.c/h：计算时间辅助文件，实现了计算两个时间的差、设定时间等函数
5. transfer.c/h:传送和文件处理的函数封装文件，实现了传送过程中对接收数据、文件的处理函数，以及发送数据包函数
6. udpPacket.c/h:数据包类型的定义封装文件，实现了不同类型数据包的生成函数以及网络、本地类型转换的辅助函数

对于 project 给的文件，除了 peer.c 文件，其他没有进行修改，所以只列出了新加入的一下文件及其用途。

文件定义函数与作用：

peer.c:

```
1.  //处理接收到的各个包的类型对应的函数声明，传入参数为接收到的数据包以及数据发送方
2.  void handle_whohas(data_packet_t *pkt, bt_peer_t *peer);
3.  void handle_ihave(data_packet_t *pkt, bt_peer_t *peer);
4.  void handle_get(data_packet_t *pkt, bt_peer_t *peer);
5.  void handle_data(data_packet_t *pkt, bt_peer_t *peer);
6.  void handle_ack(data_packet_t *pkt, bt_peer_t *peer);
7.  //根据地址获取对应的对等方
8.  bt_peer_t *get_peer(struct sockaddr_in addr);
9.  //用于记录窗口改变大小的函数，根据传入的参数记录
10. void log_window(int conn_id, int window, int time, int ssthresh, int isslows
    tart);
11. //处理 time_out 的线程函数，创建后 sleep 1 秒，期间线程可被停止，1 秒开始处理超时
12. void handle_time_out_thread(void *arg);
```

```
//用于处理超时线程的死锁
void handle_dead_lock(void *d);
//开始计时函数，创建计时线程，启动 time out 计时
void begin_time_out(up_conn_t *up_conn);
```

queue.c/h:

```
//获取队列空间,并初始化队列
queue *make_queue();
//初始化队列
void init_queue(queue *);
//清空队列
void free_queue(queue *, int);
//将数据加入队列
void enqueue(queue *, void *);
//将数据提取出队列
void *dequeue(queue *);
```

task.c/h:

```
//根据传入的 chunk 哈希和 id 用于初始化 chunk
chunk_t *make_chunk(int id, const uint8_t *sha1);
//释放 chunk
void free_chunk(chunk_t *chunk);
//初始化任务池
task_t *init_task(const char *, const char *, int);
//根据 outputfile 和 getchunkfile 初始化任务
int check_task(task_t *task);
//检测到当前任务未完成,继续下载任务
void continue_task(task_t *task, down_pool_t *pool, int sock);
//结束下载任务
task_t *finish_task(task_t *task);
//初始化上传池
void init_up_pool(up_pool_t *pool, int max_conn);
//根据对等方信息从上传池获取上传连接
up_conn_t *get_up_conn(up_pool_t *, bt_peer_t *);
//根据传入对等方地址和数据包数组的初始化上传连接并加入上传池
up_conn_t *add_to_up_pool(up_pool_t *pool, bt_peer_t *peer, data_packet_t **
pkts);
//根据对等方信息将上传连接从上传池移除
void remove_from_up_pool(up_pool_t *pool, bt_peer_t *peer);
//根据传入对等方地址和数据包数组的初始化上传连接
up_conn_t *make_up_conn(bt_peer_t *peer, data_packet_t **pkts);
//初始化下载池
void init_down_pool(down_pool_t *pool, int max_conn);
```

```
//根据传入对等方找到对应的下载连接
down_conn_t *get_down_conn(down_pool_t *, bt_peer_t *);
//根据传入对等方地址和数据包数组的初始化下载连接并加入下载池
down_conn_t *add_to_down_pool(down_pool_t *pool, bt_peer_t *peer, chunk_t *chunk);
//根据对等方信息移除相应的下载连接
void remove_from_down_pool(down_pool_t *pool, bt_peer_t *peer);
//根据传入对等方地址和 chunk 的初始化下载连接
down_conn_t *make_down_conn(bt_peer_t *peer, chunk_t *chunk);
//根据哈希值找到任务中对应的 chunk,将传入的对等方加入其下载源,以更新可用对等方
void available_peer(task_t *task, uint8_t *sha1, bt_peer_t *peer);
//根据传入的 ihave 的哈希值,选择一个 chunk 进行下载
chunk_t *choose_chunk(task_t *task, queue *chunks, bt_peer_t *peer);
//清空失效的节点
int remove_stalled_chunks(down_pool_t *pool);
//清除未回应的对等方
void remove_unack_peers(up_pool_t *pool, int sock);
//寻找 peer 队列中是否存在这个 peer
int check_peers(queue *peers, bt_peer_t *peer);
```

timer.c/h：

```
//对传入的时间赋值当前时间
void timer_start(struct timeval *);
//计算两个时间的差值，返回为毫秒
int timer_diff(struct timeval *starter, struct timeval *now);
//计算传入时间与当前时间的差值
int timer_diff_now(struct timeval *starter);
```

transfer.c/h:

```
//根据传入的 get_chunk_file 生成需要发送的 whohas 包队列
queue *init_whohas_packet_queue(const char *);
//根据传入的哈希值队列生成 ihave 包队列
queue *init_ihave_packet_queue(queue *);
//根据传入的哈希值寻找对应的 chunk,并根据 chunk 生成 data 包
//由于每个 chunk 有 512k,数据包只能有 1500 字节,所以每次读取 1024 字节生成一个 data 包,总共生成 512 包
data_packet_t **init_data_packet_array(uint8_t *sha);
//读取 chunkfile,将内容转化为二进制哈希值的队列
//此处 chunkfile 为 getchunkfile 和 haschunkfile,
queue *chunk_file_to_queue(const char *chunk_file);
//将 haschunkfile 内容转为队列
void init_has_chunks(const char *);
```

```
//根据传入的 whohas 队列,寻找本地拥有其中的某些项,并返回
//用于处理收到 whohas 包时找到本地拥有的 chunk
queue *which_i_have(queue *);
//根据传入的数据生成哈希值队列
//用于处理 whohas 包和 ihave 包的数据
queue *data2chunks_queue(void *);
//根据传入的哈希值队列,生成对应的包队列
//生成的包类型为 whohas 和 ihave 类型,根据传入的参数决定哪种类型
queue *init_chunk_packet_queue(queue *chunks, data_packet_t *(*make_chunk_packet)(short, void *));
//向特定的对等方发送 ihava 包,将队列的包全部发送
void send_pkts(int, struct sockaddr *, queue *);
//向所有对等方发送队列中 whohas 包,用于处理 get 命令时使用发送 whohas 包
void send_whohas(int sock, queue *whohas_pkts);
//向特定对等方发送数据包
void send_data_packets(up_conn_t *, int, struct sockaddr *);
//发送包的接口,发送所以包统一调用该接口,该接口调用 spiffy_send 发送数据
void send_packet(int, data_packet_t *, struct sockaddr *);
```

udpPacket.c/h:

```
//根据传入数据初始化数据包
void init_packet(data_packet_t *, char, short, u_int, u_int, char *);
//根据传入数据创建数据包,初始化数据包
data_packet_t *make_packet(char, short, u_int, u_int, char *);
void free_packet(data_packet_t *);
//创建不同类型的数据包
data_packet_t *make_whohas_packet(short, void *);
data_packet_t *make_ihave_packet(short, void *);
data_packet_t *make_get_packet(short, char *);
data_packet_t *make_data_packet(short, uint, uint, char *);
data_packet_t *make_ack_packet(uint, uint);
//将数据包头部转为网络格式
void host2net(data_packet_t *);
//将数据包头部转为本地格式
void net2host(data_packet_t *);
//判断数据包是否合法,不合法返回-1,合法返回数据包类型
int packet_is_legal(void *);
```

数据结构

queue.h:

```c
//队列中的节点
typedef struct node {
    void *data;//节点数据
    struct node *next;//下一个节点
} node;
//队列,用于存储过程中的各类链表数据
typedef struct queue {
    node *head;//队列头部
    node *tail;//队列尾部
    int n;//队列节点数量
} queue;
```

用链表实现的队列， 可以储存变长的同一类型数据， 以先进先出的存放和读取数据。项目中大概所以类型存储都用的的队列

udpPacket.h:

```c
//数据包头结构
typedef struct header_s {
    short magicnum;//magicnum
    char version;//版本号
    char packet_type;//数据包类型
    short header_len;//包头部长度
    short packet_len;//包数据长度
    u_int seq_num;//序列号
    u_int ack_num;//确认号
} header_t;
//数据包结构,包括头部与数据
typedef struct data_packet {
    header_t header;//数据包头部
    char data[DATALEN];//包所包含的数据
} data_packet_t;
```

数据包结构，udp 传送过程用到的数据包，包含头部与数据段，头部也是一个结构体，具体结构如上所示。

task.h:

```c
static int up_conn_id = 0;
//chunk 结构,用于存储需要下载的 chunk 的状态
typedef struct chunk_s {
    int id;
    uint8_t sha1[SHA1_HASH_SIZE];//chunk 的哈希值
```

```
6.      int flag;//是否下载完成
7.      int inuse;//是否在下载
8.      int num;//该 chunk 可以从多少个 peer 获取
9.      char *data;//存储 chunk 的数据
10.     queue *peers;//拥有该 chunk 的 peer
11. } chunk_t;
12. //下载连接的结构体
13. typedef struct down_conn_s {
14.     uint expect_seq;//下一个需要收到的数据包的序列号
15.     chunk_t *chunk;//该连接对应下载 chunk
16.     int pos;//当前数据包下载的位置指针
17.     struct timeval timer;
18.     bt_peer_t *sender;//下载源
19. } down_conn_t;
20. //上传连接的结构体
21. typedef struct up_conn_s {
22.     int up_id;//当前上传连接的 id，每一个连接加 1
23.     int ssthresh;//当前上传连接的慢启动阈值
24.     int is_slow_start;//记录是否是慢启动状态
25.     int receive_ack_during_avoid;//记录拥塞避免期间收到的 ack 数量
26.     pthread_t pid;//管理这个连接的 time out 的线程 pid
27.     int expect_ack;//下一个期待的 ack
28.     int last_sent;//上一个发送的包的序列号
29.     int available;//可以发送的包的最大序列号
30.     int duplicate;//记录收到重复 ack 数量
31.     int cwnd;//接收方窗口大小
32.     data_packet_t **pkts;//需要上传的数据包数组
33.     struct timeval timer;//记录时间
34.     bt_peer_t *receiver;//接收数据方
35. } up_conn_t;
36. //下载池,保存了当前的下载连接
37. typedef struct down_pool_s {
38.     down_conn_t **conns;//正在使用的下载连接数组
39.     int conn;//当前下载池数量
40.     int max_conn;//最大下载数量
41. } down_pool_t;
42. //上传池
43. typedef struct up_pool_s {
44.     up_conn_t **conns;//正在使用的上传连接数组
45.     int conn;//当前上传池数量
46.     int max_conn;//最大上传数量
47. } up_pool_t;
48. //当前对等方下载任务
49. typedef struct task_s {
```

```
50.     int chunk_num;//已经下载的 chunk 数量
51.     int need_num;//需要下载的 chunk 数量
52.     queue *chunks;//需要下载的 chunk 队列
53.     int max_conn;
54.     char output_file[BT_FILENAME_LEN];//下载的数据存储到该路径
55.     struct timeval timer;
56. } task_t;
```

每个 chunk_t 对应着一个 chunk，保存当前 chunk 的状态，是否下载、结束以及下载的数据等等，

每个 conn_t 都是一个与其他对等方的连接，保存着上传或者下载的信息，pool 则保存着所有的 conn_t；

task_t 是下载任务的维持者，按照 get_chunk_file 初始化之后，保持着需要下载的任务即 chunk，直到下载任务完成后才会释放。

peer.c

```
1.  typedef struct time_out_arg {
2.      int seq_num;
3.      int id;
4.  } out_args;
```

存储线程参数的结构，用于向管理超时线程的函数传递参数。

程序逻辑

1. peer 程序启动后，进行一系列初始化操作后，进入 peer_run 函数，阻塞等待用户的输入或等待其他对等方发送的消息。
2. 接收到消息后有两种情况，来自用户的输入，进行步骤 3 处理，来自其他对等方的消息，进行步骤 4 处理
3. **处理用户的输入：**调用 bt_parse.c 的函数序列化用户输入，并处理用户的 get 指令：根据用户的 get_chunk_file 生成 whohas 的数据包，并把 whohas 数据包发送给所有对等方，等待其他对等方发送消息。
4. **处理对等方消息：**对等方消息由于类型的不同，有不同的处理函数，根据类型到达对应的处理步骤：
   whohas 类型：　步骤 5
   ihave 类型：　步骤 6
   get 类型：　步骤 7
   data 类型：　步骤 8
   ack 类型：　步骤 9

5. **处理 whohas 类型包：**接收到 whohas 数据包，首先获取该数据包发送方 A，并获取数据包中所有的哈希值，然后将哈希值和本地 has_chunk_file 的哈希值进行比较，寻找到所有本地与数据包中相同的哈希值。如果存在相同的

值，将这些值封装生成 ihava 数据包，然后将 ihava 数据包发还给 A。如果不存在相同的值，则不采取操作

6. **处理 ihava 类型包：**在向其他对等方发送 whohas 包后，其他对等方会发回 ihava 包，说明他们所拥有的 chunk。接收到 ihava 包后，处理 ihava 包的数据，根据 ihave 包的哈希值在 task 中为每个 chunk 保存可用于下载的对等方，然后先挑一个 chunk，根据其哈希值创建 get 类型包，发送给对应的一个对等方，建立连接，进行下载该 chunk，并加入下载池。如果接收到不同对等方发过来的 ihava 包，可以建立不同连接，同时下载。
7. **处理 get 类型包：**接收到 get 类型包时，说明该对等方 A 需要下载一个本地拥有的 chunk。获取 get 包中的哈希值，根据哈希值在 master_file 中查找其该 chunk 的 id，根据 id 获取 chunk 的数据，构建 512 个 data 数据包（由于 udp 包大小有限，每个 chunk 的大小固定为 512k，所以每个构建的 udp 包包含 1k 的数据），然后根据当前窗口大小，将数据包分步发送给对等方 A，同时启动 time out 计时器
8. **处理 data 类型包：**data 类型包包含所需要的下载数据。如果该包的发送方不是我们所连接的发送方，说明是一个发错的包，不采取操作。如果是连接的发送方，先判断 seqnum 是不是我们所期望的 seqnum，如果是，获取对应的 chunk，然后将获取的数据存入 chunk 的缓存区中，并发送对应的 ack 包进行确认收到；如果 seqnum 达到最大，则表示下载完成，结束这个 chunk 的下载任务。如果不是我们需要的 seqnum，则忽视这个包，并且发送一个重复的 ack 表示没有收到正确的包。
9. **处理 ack 类型包：**一样进行判断是不是与我们连接的对等方，如果不是，忽略。如果是，判断 acknum。acknum 分为三种情况：

   1.判断 acknum 是否达到最大，如果是，说明这个 chunk 以及上传完毕，结束该上传任务，停止计时器

   2. 如果 acknum 大于等于我们所期望的 acknum，那么就说明前面的包都已经成功发送（累计确认）。此时还有数据没有上传，则增加对应的变量，并根据当前的状态（慢启动或者拥塞避免）来改变窗口的大小，然后按照当前窗口容量继续发送剩下的数据包，并重启计时器。

   3.如果 acknum 等于我们所期待的 acknum － 1，那么重复记录加一，如果重复记录达到 3，说明丢包，马上改变窗口大小为 1，改变阈值，进入快速重传，根据窗口重发没有确认的第一个包，重启计时器。
10. 处理结束之后，peer 会继续在 peer_run 函数里阻塞等待其他对等方的消息或者用户的输入。

关于超时与拥塞控制

拥塞控制主要以下列几种情况完成：

1. **慢启动状态：**初始窗口为 1，慢启动状态下，每次收到一个 ack，窗口大小增加 1，当达到阈值之后，进入拥塞避免状态
2. **拥塞避免状态：**窗口大小达到阈值之后的状态，此时不再快速增加窗口大小，而是每次在收到 ack 后累加收到 ack 与期望的 ack 差值，当累加值大于等于当前窗口大小时，说明至少经过一个 RTT，则窗口加 1
3. **快速重传：**将阈值设置为当前窗口大小的一半（至少为 2），并且将窗口大小改为 1，然后按照窗口大小重发未被确认的包，恢复慢启动状态。在收到重复

ack 3 次或者超时后，都会进行快速重传，并重启计时器

4. **重复 ack**：收到的 ack 等于期望的 ack-1 的时候代表重复 ack，当连续 3 次收到重复 ack 时，则表示可能丢包了，进行快速重传
5. **超时**：每次收到成功的 ack 后或者快速重传后，会重启计时器，开启计时线程，计时线程会 sleep 1 秒，期间可以被取消，如果计时线程没有被取消，那么就表示超时，超时马上进行快速重传。

关于并发运行

为了满足可以同时上传或者下载，采用了上传与下载池的操作，每次接收到上传或者下载请求，会建立对应的上传或者下载链接，每条链接对应一个 peer，连接记录与该 peer 交互的状态，例如上传了多少文件等信息

每次上传或者下载都会从上传池或者下载池中根据 peer 找到对应的连接，经过该连接来进行操作。由于每个连接互不干扰，所以每次接收到一个包，只需要寻找到对应的连接就可以进行操作，实现了没有采用多线程的并发操作。

开发过程遇到的问题

1. 实现拥塞避免过程中发现程序无法进入拥塞避免状态
   经过检查发现，原先采用的拥塞避免的方法是检测上一次改变窗口的时间与这次收到 ack 的时间相差是否达到一秒。由于 1 秒时间太长，极易发生重复 ack 等问题。后来修改实现，每次发送当前窗口大小的分组数量后，只要这些分组都被确认，就说明经过了一个 RTT，此时增加一个窗口大小。
2. 由于超时存在，不知道怎么解决超时问题
   最后经过思考，采用多线程解决问题，每次重启计时器的时候，就撤销上一个计时器（也就是上一个线程），然后创建新的线程，并 sleep 1 秒。1 秒内如果线程被取消，就说明没有超时；1 秒后，线程苏醒，代表超时，执行超时操作即可

程序运行结果图：

图片也放在文档的同级目录下

cp1 测试：

```
chen@chentao:~/Bing/network/Project1/Starter Code/cp1$ ruby checkpoint1.rb
starting SPIFFY on port 15441
starting tests
Spiffy local stuff:  0100007f:1111
15-441 PROJECT 2 PEER

chunk-file:     C.chunks
has-chunk-file: B.chunks
max-conn:       4
peer-identity:  2
peer-list-file: nodes.map
  peer 2: 127.0.0.1:2222
  peer 1: 127.0.0.1:1111
Spiffy local stuff:  00000000:2222
Spiffy setup complete.  127.0.0.1:15441
Delete this line after testing.
Spiffy setup complete.  127.0.0.1:15441
Delete this line after testing.
Listening on 15441...
PROCESS GET SKELETON CODE CALLED.  Fill me in!  (A.chunks, silly.tar)
verifying packet sanity
########### Test 1 Passed! ###########
15-441 PROJECT 2 PEER

chunk-file:     C.chunks
has-chunk-file: A.chunks
max-conn:       4
peer-identity:  2
peer-list-file: nodes.map
  peer 2: 127.0.0.1:2222
  peer 1: 127.0.0.1:1111
Spiffy local stuff:  00000000:2222
Spiffy setup complete.  127.0.0.1:15441
```

```
peer-list-file: nodes.map
  peer 2: 127.0.0.1:2222
  peer 1: 127.0.0.1:1111
Spiffy local stuff:  00000000:2222
Spiffy setup complete.  127.0.0.1:15441
Delete this line after testing.
Spiffy local stuff:  0100007f:1111
Spiffy setup complete.  127.0.0.1:15441
Delete this line after testing.
creating Control packet of type 0 with 2 chunks
verifying packet sanity
########### Test 2 Passed! ###########
15-441 PROJECT 2 PEER

chunk-file:     C.chunks
has-chunk-file: A.chunks
max-conn:       4
peer-identity:  2
peer-list-file: nodes.map
  peer 2: 127.0.0.1:2222
  peer 1: 127.0.0.1:1111
Spiffy local stuff:  00000000:2222
Spiffy setup complete.  127.0.0.1:15441
Delete this line after testing.
Spiffy local stuff:  0100007f:1111
Spiffy setup complete.  127.0.0.1:15441
Delete this line after testing.
creating Control packet of type 0 with 2 chunks
Good, went ten seconds without receiving IHAVE
########### Test 3 Passed! ###########
done with tests
killing spiffy
```

cp2 测试：

```
chen@chentao:~/Bing/network/Project1/Starter Code/cp2$ ruby checkpoint2.rb
starting SPIFFY on port 15441
starting tests
Spiffy local stuff:  00000000:2222
Spiffy setup complete.  127.0.0.1:15441
Delete this line after testing.
Spiffy local stuff:  0100007f:1111
Spiffy setup complete.  127.0.0.1:15441
Delete this line after testing.
Listening on 15441...
PROCESS GET SKELETON CODE CALLED.  Fill me in!  (test1.chunks, test1.tar)
id = 0, hash = 6acfce07d222d400ce900d918306c6664501b33f
id = 1, hash = af91dbd47aac60e980e0369df68271ca52de11a8
******** ref_peer exiting after 2 connections, as specified by -x flag ********
############ Test 1 Passed! ############
Spiffy local stuff:  00000000:1111
Spiffy setup complete.  127.0.0.1:15441
Delete this line after testing.
Spiffy local stuff:  0100007f:2222
Spiffy setup complete.  127.0.0.1:15441
Delete this line after testing.
id = 0, hash = 3b9f916bbf59021ab781c9f2456df90a0102079f
id = 0, hash = 3b9f916bbf59021ab781c9f2456df90a0102079f
id = 0, hash = 3b9f916bbf59021ab781c9f2456df90a0102079f
id = 1, hash = 78585121ee33fbb6666bc4bf1a8498c04b2758e8
id = 1, hash = 78585121ee33fbb6666bc4bf1a8498c04b2758e8
id = 1, hash = 78585121ee33fbb6666bc4bf1a8498c04b2758e8
GOT test2.tar
********* ref_peer exiting after 2 connections, as specified by -x flag ***********
############ Test 2 Passed! ############
done with tests
killing spiffy
```

cp3 测试：

```
Dropping packet seq = 348, ack = 0 len = 1432 (from,to) = (127.0.0.1:1111,127.0.0.1:2222) at time = 97404
Dropping packet seq = 349, ack = 0 len = 1432 (from,to) = (127.0.0.1:1111,127.0.0.1:2222) at time = 98409
Dropping packet seq = 353, ack = 0 len = 1432 (from,to) = (127.0.0.1:1111,127.0.0.1:2222) at time = 99425
Dropping packet seq = 0, ack = 352 len = 32 (from,to) = (127.0.0.1:2222,127.0.0.1:1111) at time = 99429
Dropping packet seq = 354, ack = 0 len = 1432 (from,to) = (127.0.0.1:1111,127.0.0.1:2222) at time = 100441
Dropping packet seq = 359, ack = 0 len = 1432 (from,to) = (127.0.0.1:1111,127.0.0.1:2222) at time = 101455
Dropping packet seq = 360, ack = 0 len = 1432 (from,to) = (127.0.0.1:1111,127.0.0.1:2222) at time = 101461
Dropping packet seq = 362, ack = 0 len = 1432 (from,to) = (127.0.0.1:1111,127.0.0.1:2222) at time = 102476
Dropping packet seq = 363, ack = 0 len = 1432 (from,to) = (127.0.0.1:1111,127.0.0.1:2222) at time = 102480
Dropping packet seq = 366, ack = 0 len = 1432 (from,to) = (127.0.0.1:1111,127.0.0.1:2222) at time = 103496
******** ref_peer exiting after 2 connections, as specified by -x flag ********
############ Test 1 Passed! ############
15-441 PROJECT 2 PEER

chunk-file:     C.chunks
has-chunk-file: B.chunks
max-conn:       4
peer-identity:  1
peer-list-file: nodes.map
  peer 2: 127.0.0.1:2222
  peer 1: 127.0.0.1:1111
Spiffy local stuff:  00000000:1111
Spiffy local stuff:  0100007f:2222
Dropping packet seq = 500, ack = 0 len = 1056 (from,to) = (127.0.0.1:1111,127.0.0.1:2222) at time = 170958
Dropping packet seq = 502, ack = 0 len = 1056 (from,to) = (127.0.0.1:1111,127.0.0.1:2222) at time = 171977
Dropping packet seq = 0, ack = 502 len = 32 (from,to) = (127.0.0.1:2222,127.0.0.1:1111) at time = 171978
Dropping packet seq = 507, ack = 0 len = 1056 (from,to) = (127.0.0.1:1111,127.0.0.1:2222) at time = 171988
Dropping packet seq = 0, ack = 505 len = 32 (from,to) = (127.0.0.1:2222,127.0.0.1:1111) at time = 171989
Dropping packet seq = 0, ack = 506 len = 32 (from,to) = (127.0.0.1:2222,127.0.0.1:1111) at time = 172006
Dropping packet seq = 507, ack = 0 len = 1056 (from,to) = (127.0.0.1:1111,127.0.0.1:2222) at time = 172998
Dropping packet seq = 509, ack = 0 len = 1056 (from,to) = (127.0.0.1:1111,127.0.0.1:2222) at time = 174013
Dropping packet seq = 0, ack = 510 len = 32 (from,to) = (127.0.0.1:2222,127.0.0.1:1111) at time = 174020
Successfully Downloaded: id = 1, hash = 78585121ee33fbb6666bc4bf1a8498c04b2758e8
GOT test2.tar
********* ref_peer exiting after 2 connections, as specified by -x flag ***********
############ Test 2 Passed! ############
done with tests
killing spiffy
```

并发测试

```
chen@chentao:~/chen/computernetwork/Starter Code/concurrenttest$ ruby concurrenttest.rb
starting SPIFFY on port 15441
drop rate = 0.00

droptimes str = 60
key = 127.0.0.1:1111,127.0.0.1:2222 droptimes = 60
Listening on 15441...
starting test
Spiffy local stuff:  0100007f:1111
Spiffy setup complete.  127.0.0.1:15441
Delete this line after testing.
Spiffy local stuff:  0100007f:2222
Spiffy setup complete.  127.0.0.1:15441
Delete this line after testing.
15-441 PROJECT 2 PEER

chunk-file:     ./C.chunks
has-chunk-file: B.chunks
max-conn:       4
peer-identity:  3
peer-list-file: nodes.map
  peer 3: 127.0.0.1:3333
  peer 2: 127.0.0.1:2222
  peer 1: 127.0.0.1:1111
Spiffy local stuff:  00000000:3333
Spiffy setup complete.  127.0.0.1:15441
Delete this line after testing.
PROCESS GET SKELETON CODE CALLED.  Fill me in!  (test1.chunks, test1.tar)
starting flow 127.0.0.1:3333,127.0.0.1:2222 at 1545310730749.14
Peer 1 received WHO_HAS with 2 hashes
id = -1, hash = 6acfce07d222d400ce900d918306c6664501b33f
id = -1, hash = af91dbd47aac60e980e0369df68271ca52de11a8
found 2 matches:
id = 0, hash = 6acfce07d222d400ce900d918306c6664501b33f
id = 1, hash = af91dbd47aac60e980e0369df68271ca52de11a8
creating Control packet of type 1 with 2 chunks
Peer 1 Sending IHAVE of size 60 to 0.0.0.0:3333
Peer 2 received WHO_HAS with 2 hashes
id = -1, hash = 6acfce07d222d400ce900d918306c6664501b33f
id = -1, hash = af91dbd47aac60e980e0369df68271ca52de11a8
found 2 matches:
id = 0, hash = 6acfce07d222d400ce900d918306c6664501b33f
id = 1, hash = af91dbd47aac60e980e0369df68271ca52de11a8
creating Control packet of type 1 with 2 chunks
Peer 2 Sending IHAVE of size 60 to 0.0.0.0:3333
Peer 1 received GET from peer 3:
id = 0, hash = 6acfce07d222d400ce900d918306c6664501b33f
Data Transfer May take a few minutes
Peer 2 received GET from peer 3:
id = 1, hash = af91dbd47aac60e980e0369df68271ca52de11a8
Data Transfer May take a few minutes
******** ref_peer exiting after 1 connections, as specified by -x flag ********
******** ref_peer exiting after 1 connections, as specified by -x flag ********
########### Test 1 Data is Correct ###########
killing spiffy after test1
done with test
```