P2P-BitTorrtent 说明文档

一． 代码概述

该 Project 的代码文件和代码实现的概述，请看同级目录下的“Readme.md”。该文档，即“Design.pdf”，主要描述该 Project 的总体设计和代码的具体实现。

二． 具体设计

(1) 文件传输部分

(I)可靠传输

接收方采用 GBN（回退 N 步协议）累积确认的方式，发送方将“3 次冗余 ACK”和“超时”（通过 SIGALRM 信号来实现定时器，在该 Project 中，我将定时的时长固定为 2 秒）均定义为丢包事件的发生，从而重新发送相应的分组。为了模拟发送方和接收方的连接状态，定义结构 up_conn_s 和 down_conn_s。通过 up_conn_s 和 down_conn_s 管理对应连接的状态，具体的 up_conn_s 和 down_conn_s 结构的设计在“Readme.md”中有详细说明。

下载流程：当一个 peer 接收到来自用户命令行的 GET 任务请求时，先向其它所有的 peer(s)发送 WHOHAS 分组（考虑到要下载的 chunk 可能很多，所以可能要分多次发送）。当接收到 IHAVE 分组时，通过 handle_ihave 方法建立下载连接并加入 down_pool 中，更新 GET 任务中对应要下载的 chunk 的状态，发送 GET 分组。接收到来自发送方的 DATA 分组后，通过 handle_data 回复对应的 ACK 分组，并缓存数据，更新任务状态。该 chunk 的下载任务结束后，从 down_pool 中移除下载连接，接收方通过 shahash 对比下载的 chunk 数据是否正确，若正确，则将数据写入该 chunk 在 GET 任务中对应的缓存。

上传流程：当一个 peer 收到 WHOHAS 分组时，通过 handle_whohas 回复对应的 IHAVE 分组（如果有对应的 chunk(s)）。当接收到 GET 分组时，通过 handle_get，建立上传连接并加入 up_pool 中，发送 DATA 分组，开始计时。接收到 ACK 分组时，通过 handle_ack，①正常 ACK，更新窗口状态，继续发送分组，开始计时；②冗余 ACK，up_conn->dup_times 加一，若 dup_times >= 3, 认为丢包事件发生，更新窗口状态，重发丢失的分组。若发生了超时情况，通过 handle_timeout 方法，重发对应分组并更新窗口状态。上传任务结束后，从 up_pool 中移除对应的上传连接。

(II)同时传输

为了实现同时传输，定义结构 up_pool_s 和 down_pool_s 来管理对应 peer 所有的上传和下载连接。

为了更好地利用 p2p 中所有的对等方，对于 GET 任务（即 task_get_s）中的 chunk(s)下载任务，用 bt_peer_t **providers 管理 chunk(s)的提供方（即发送方），用 int *status 来管理所有要下载的 chunk(s)的状态。status = 0 表示对应的 chunk 还未开始下载；status = 1 表示对应的 chunk 已经下载完成；status = 2 表示对应的 chunk 正在下载。

所以当出现下述情况时，(client) peer 可以更高效地利用其它所有(server) peers 的资源。例如，peer A 要下载 chunk1 和 chunk2，peer B 和 peer C 都有 chunk1 和 chunk2。当 peer A 群发“WHOHAS”时，peer B 和 peer C 都会回复 IHAVE chunk1&chunk2。当其中一个“IHAVE”到达时（假设 peer B 的 IHAVE 先到达），peer A 可以先从 peer B 下载 chunk1（这时 chunk1 的 status 为 2，chunk2 的 status 仍为 0）。然后，当 peer C 的“IHAVE”到达时，peer A 就可以从 peer C 获取 chunk2，更好地利用了所有 peers 的资源，也提高下载的效率。

(2) 拥塞控制部分

采用“TCP Tahoe”策略实现拥塞控制。慢启动：初始的 cwnd 为 1，ssthresh 为 64，在未发生拥塞（丢包事件）或 cwnd 未达到 ssthresh 时，每收到新的确认 ACK，cwnd 加一。超时和快速重传：当出现拥塞（3 个冗余 ACK 或超时），ssthresh 设为 max(cwnd/2, 2)，cwnd 设为 1，然后继续慢启动。拥塞避免：当 cwnd 达到 ssthresh 时，直到上一次 cwnd 内的分组全部被确认接收成功后，cwnd 才加一，其它时候 cwnd 保持不变。

当 peer 为发送方时，每一次发送分组或接收到 ACK 时，程序都将对应发送窗口的 cwnd 值或接收到的 ACK 值记录到文件“problem2-peer.txt”中。

格式说明：①对于窗口值 cwnd 记录，第一列表示连接的 ID（格式为 conn-senderID-receiverID，例如“conn1-2”表示 peer1 发送数据给 peer2 的上传连接），第二列表示从上传连接创建到发送该分组所用的时间（毫秒），第三列表示 cwnd 的值。②对于 ACK，用类似“receive ACK 1”的格式记录。

某一次实验的“problem2-peer.txt”文件的部分截图（即收到的 ACK 记录和拥塞控制窗口变化的记录），如下图所示。

```
conn1-2    1    1
receive ACK 1
conn1-2    111    2
receive ACK 2
conn1-2    182    3
receive ACK 3
conn1-2    252    4
receive ACK 4
conn1-2    332    5
receive ACK 5
conn1-2    1246    6
receive ACK 6
conn1-2    2091    7
receive ACK 7
conn1-2    2147    8
receive ACK 9
conn1-2    2212    10
```

（图一. 慢启动）

```
receive ACK 9
conn1-2    2212    10
receive ACK 9
receive ACK 9
receive ACK 9
conn1-2    2412    1
```

（图二. 丢包）

```
conn1-2    3252    9
receive ACK 26
conn1-2    3300    10
receive ACK 28
conn1-2    3364    10
receive ACK 29
conn1-2    3409    10
receive ACK 30
conn1-2    3490    10
receive ACK 31
conn1-2    3581    10
receive ACK 32
conn1-2    3625    10
receive ACK 33
conn1-2    3724    10
receive ACK 34
conn1-2    3794    10
receive ACK 35
conn1-2    3821    10
receive ACK 36
conn1-2    3893    11
receive ACK 37
conn1-2    3994    11
receive ACK 38
conn1-2    4100    11
receive ACK 39
conn1-2    4202    11
receive ACK 40
conn1-2    4304    11
```

（图三. 慢启动+拥塞避免）

三. 文件传输和拥塞控制部分的代码实现

(1) task_get.[h | c]

void init_task(char *chunk_file, char *out_file);
根据<get-chunk-file> chunk_file 和输出文件名 out_file，初始化 GET 任务。
char *update_provider(list_t *chunk_hash_list, bt_peer_t *peer)
对于 chunk_hash_list 中的每个 chunk hash，如果 task_get 中对应的 status 为 0，则将对应的提供方更新为 peer，再在所有 status 为 0，provider 为 peer 的 chunks 中找一个，作为接下来要下载的 chunk。并返回。
void add_and_check_data(char *hash, char *data);
添加一个已下载的 chunk 数据到任务 task_get 中，并通过 shahash 判断数据是否有错误。
int is_task_finish();
通过所有 chunk 的 status 是否都为 1，来判断任务是否都已经完成。
void write_data();
将任务中缓存的所有 chunk 数据，写入到任务所对应的输出文件中。

(2) handler.[h | c]

void pkt_ntoh(packet_t *packet);
将收到的分组转换为主机字节顺序
int parse_type(packet_t *packet);
解析对应分组的类型
packet_t *new_pkt(unsigned char type, unsigned short packet_len, unsigned int seq_num, unsigned int ack_num, char *payload);
根据参数，生成一个对应的分组并返回。
packet_t *handle_whohas(packet_t *pkt_whohas);
用来处理接收到的 WHOHAS 分组，遍历该 peer 所拥有的所有 chunk(s)，得到符合要求的 chunk(s) hash，并打包成 IHAVE 分组返回；若没有符合要求的 chunk 则返回 NULL。
list_t *new_whohas_pkt(); 根据已经初始化的 GET 任务，生成一个 WHOHAS 分组的链表。（一个 GET 任务可能有很多需要请求的 chunks，又考虑到一个分组最多容纳 1500 字节，所以可能需要多个 WHOHAS 分组来向其它 peers 发送 WHOHAS。）
packet_t *handle_ihave(packet_t *pkt, bt_peer_t *peer);
在下载连接池未满时，根据接收到的 IHAVE 分组，先调用 update_provider(chunks, peer)，更新所有 chunks 的 status 和 provider，再通过返回的要下载的 chunk hash 创建下载连接，生成对应的 GET 分组请求并返回。
void handle_get(int sock, packet_t *pkt, bt_peer_t *peer);
在上传连接池未满时，根据收到的 GET 分组，从 tracker（管理所有的 chunks）中获取该 GET 分组请求所需要的 chunk 所对应的数据（共 512k 字节）。将数据拆分为 512 个 DATA 分组，这 512 个 DATA 分组便是发送方所需要发送的所有分组。根据这些要发送的 DATA 分组和 peer（接收方），创建一个上传连接，并开始发送 DATA 分组并计时。
void handle_data(int sock, packet_t *pkt, bt_peer_t *peer);
接收方收到发送方 peer 发送的数据：若该数据的 seq 是该下载连接所期待的（即 seq == down_conn->next_ack），则将数据缓存到下载连接的 chunk_buf 中并回复一个 ACK = seq 的 ACK 分组；否则回复一个冗余的 ACK（即 ACK = down_conn->next_ack - 1）。 然后，判断该下载连接管理的 chunk 的所有(512k)数据是否都已经接收到，如果都已经收到，则检查该 chunk 数据是否正确，更新该 chunk 在 GET 任务中对应的 status，再将获取的数据存入到 GET 任务对应的缓存中，移除该下载连接。最后判断 GET 任务中所有的 chunk 数据请求是否都已经正确下载完成，如果完成，则将所有的 chunk 数据按顺序写入到输出文件中；否则继续为 status ==0 且存在 provider 的 chunk 请求数据，建立下载连接并发送 GET 分组。
void handle_ack(int sock, packet_t *pkt, bt_peer_t *peer);
根据可靠传输和拥塞控制机制。①ACK == 512，因为在 handle_get 时，发送方将 512k 数据拆分为了 512 个 DATA 分组，所以 ACK == 512 表示所有的数据都已经收到，可以停止计时并将上传连接移除；②ACK > 上一次接收到的 last_ack，根据上文介绍的拥塞控制策略，判断所处阶段（慢启动还是拥塞避

免），更新发送窗口的状态，继续发送 DATA 分组并重新开始计时；③ACK 冗余，则记录冗余 ack 的 dup_times++。当 dup_times >= 3 时，则意味着数据包丢失了，to_send 回退到 last_ack 对应的位置，更新窗口状态和阈值，重新发送丢失的 DATA 分组并开始计时。
void handle_timeout()；
当发生超时时，根据拥塞控制策略，更新窗口状态和阈值，重发丢失的 DATA 分组并重新计时。