Design

By 黄子豪 18302010034

TA_READ_ME_PLEASE!

由于linux文件路径的不同，我修改了三个cp文件夹中的checkpoint.ruby以及concurrent文件夹中的concurrent.ruby，修改的地方是：在四个ruby文件中的靠近末尾的位置，

```
spiffy_pid = fork do
    exec("perl ./hupsim.pl -m topo.map -n nodes.map -p #{spiffy_port} -v 0")
end
```

添加了"perl"，请TA务必使用我的整个Start Code文件夹（包括里面的cp*和concurrent文件夹，而不仅仅是start_code）进行测试，或者是修改上述路径。

整体设计思路

在peer.c中的main函数中，首先会进行一些必要的准备工作例如申请内存、初始化down_pool等，然后执行peer_run，peer_run在构建好socket之后便进入一个循环：要么接收用户的输入，要么接收其他peer的连接请求，接下来分为文件请求者和文件提供者两个角色的peer进行讲解。

首先是文件请求者即接收方，当用户输入GET命令时，接收方向所有peer广播WHOHAS分组，然后等待其他peer回应的IHAVE分组。当IHAVE分组到达时调用函数*handler_IHAVE*，与源ip及端口号建立连接，并且考虑到需要实现同时下载，把这个连接加到连接池down_pool中。当接收到了发送方的数据报之后，调用函数*handle_data*回复ACK指令，这里对于ACK的回复遵循了GBN累计确认，当整个chunk下载完成后连接会被移除。

然后是文件提供者即发送方，发送方当收到WHOHAS分组时，调用函数*handler_whohas*，如果自己本地有这个文件则会回应IHAVE分组。当接收方确定要下载这个文件时会建立与发送方的连接，发送数据报，并且启动计时器，这里我的设计思路模仿了TCP，当接收到3个重复的ACK或者是计时器超时，则认为发生了丢包的情况，此时发送方会重发原数据报，并且把窗口起始位置调整到需要重发的包的位置。上传任务结束后，该连接从up_pool中移除。

除了上述大体的设计思路，还值得一提的是为了同时传输应用到了类似于线程池的处理方式，定义了up_pool_s和down_pool_s来管理和分配对应的上传和下载的连接，并且会采取一种较优的、与之前已分配下载任务所不同peer进行下载，例如当有多个peer拥有多个相同的所需trunks时，它们都会被加入到down_pool中，并且下载管理总是会尽可能地挑选一个当前没有下载任务的peer，从而最大化利用网络带宽。

具体代码实现

the main data structures

为了实现接收和发送文件的需求，管理down_pool和up_pool，我根据要求文档上的提示定义了一些数据结构，这里解释一些主要的数据结构。

1、接收一个传输的文件trunk：

```
typedef struct task_get_s {
     int get_num; //chunks number of this GET task
     int *status; //0: not start; 1: done; 2: processing.
     bt_peer_t **providers; //senders for each chunk
     chunk_t *get_chunks; //chunks to get
     char **chunk_data; //chunk data for each chunk
     char output[BT_FILENAME_LEN]; //output file
   } task_get_t;
```

2、为每个接收方和发送方初始化的buffer区域：

```
typedef struct chunk_s {
     int id;
     char chunk_hash[SHA1_HASH_SIZE]; //chunk hash(20 bytes) after shahash
   } chunk_t;

typedef struct chunk_buffer_s {
     char chunk_hash[SHA1_HASH_SIZE];
     char *data_buf; //chunk data of chunk_hash
   } chunk
```

3、为了实现从多个对等peer处下载或者是给多个对等peer发送trunk，定义了下载连接池和上传连接池：

```
   typedef struct down_pool_s {
     down_conn_t **conns;
     int conn_num; //current download connection number of peer
     int max_conn; //maxmium download connection number of peer
   } down_pool_t;

   typedef struct up_pool_s {
     up_conn_t **conns;
     int conn_num;
     int max_conn;
   } up_pool_t;
```

4、连接池中用到的conns结构由一下structure定义，这实现了peer之间的简单连接：

```
   typedef struct down_conn_s {
     int from_here; //used when caching data into chunk_buf->data_buf
     int next_ack; //next ack expected, range form 1 to 512
     bt_peer_t *sender;
     chunk_buffer_t *chunk_buf; //used to cache data
   } down_conn_t;

   typedef struct up_conn_s {
     int last_ack;
     int to_send; //index of packet to send, range from 0 to (512 - 1)
```

```
    int available; //index(of last available packet in sender's window) + 1
    int dup_times; //duplicate times of last_ack
    int cwnd; //(dynamic)congestion window size
    int ssthresh; //slow start threshold
    int rtt_flag; //represent the end of a rtt
    long begin_time;
    packet_t **pkts; //cache all packets to send
    bt_peer_t *receiver;
} up_conn_t;
```

the main supporting files

为了提高代码的可读性和可复用性，我把一些易分块的、可重复使用的辅助代码抽象成单个函数文件以供上层调用，例如把对WHOHAS分组、IHAVE分组的处理都抽象集中到handler.c中，以下对这些代码文件进行简单的解释。

1、packet.h

这一部分完全依照需求文档上的指导，定义了每个包的头部信息。

```
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;

typedef struct packet_s {
    header_t header;
    char data[DATALEN];
} packet_t;
```

2、store.c/h

这个文件是用来管理当前peer拥有哪些chunk信息的，当收到WHOHAS分组时，只需要查询*list_t *chunk_ihave*中是否有这个分组即可。

```
void init_tracker() {
    FILE *fd = fopen(config.chunk_file, "r");
    char info_buf[256], line_buf[256];
    fgets(line_buf, 256, fd);
    sscanf(line_buf, "%s %s", info_buf, master_file_name);
    int chunk_id;
    fgets(line_buf, 256, fd); //omit the line: (Chunks:)
    while (fgets(line_buf, 256, fd)) {
        if (sscanf(line_buf, "%d %s", &chunk_id, info_buf) != 2) {
            continue;
        }
        chunk_t *chunk = malloc(sizeof(chunk_t));
        chunk->id = chunk_id;
        hex2binary(info_buf, 2 * SHA1_HASH_SIZE, chunk->chunk_hash);
        add_node(chunk_tracker, chunk);
    }
    fclose(fd);
```

```
}

void init_chunks_ihave() {
    FILE *fd = fopen(config.has_chunk_file, "r");
    char hash_buf[256], line_buf[256];
    int chunk_id;
    while (fgets(line_buf, 256, fd)) {
        if (sscanf(line_buf, "%d %s", &chunk_id, hash_buf) != 2) {
            continue;
        }
        chunk_t *chunk = malloc(sizeof(chunk_t));
        chunk->id = chunk_id;
        hex2binary(hash_buf, 2 * SHA1_HASH_SIZE, chunk->chunk_hash);
        add_node(chunk_ihave, chunk);
    }
    fclose(fd);
}
```

3、list.c/h

底层使用list实现了一个单向链表，上层可以调用它来以滑动窗口的形式发送文件。由于是单向链表的形式，实现了*add_node*、*pop_node*函数。

```
list_t *init_list() {
    list_t *list = malloc(sizeof(list_t));
    list->node_num = 0;
    list->head = NULL;
    list->tail = NULL;
    return list;
}

void add_node(list_t *list, void *data) {
    node_t *node = malloc(sizeof(node_t));
    node->data = data;
    node->next = NULL;
    list->node_num++;
    if (list->head == NULL) {
        list->head = node;
    }
    if (list->tail != NULL) {
        list->tail->next = node;
    }
    list->tail = node;
}

void *pop_node(list_t *list) {
    if (list->node_num == 0) {
        return NULL;
    }
    node_t *node = list->head;
    void *data = node->data;
    list->head = node->next;
    list->node_num--;
    if (list->node_num == 0) {
        list->tail = NULL;
    }
    return data;
```

```
}
```

4、conn.c/h

这一部分抽象出来，为了给peer之间建立一个简单的TCP连接并且在连接池中统一地管理它们。

主要包括以下函数：

init_chunk_buffer、init_down_conn、init_down_pool、init_up_pool、init_up_conn、free_chunk_buffer：这些函数都是为了初始化/释放某个对象结构，操作类似，故不过多赘述，仅区两个作为展示。

```
chunk_buffer_t *init_chunk_buffer(char *hash) {
    chunk_buffer_t *chunk_buf = malloc(sizeof(chunk_buffer_t));
    memcpy(chunk_buf->chunk_hash, hash, SHA1_HASH_SIZE);
    chunk_buf->data_buf = malloc(BT_CHUNK_SIZE);
    return chunk_buf;
}

void free_chunk_buffer(chunk_buffer_t *chunk_buf) {
    free(chunk_buf->data_buf);
    free(chunk_buf);
}
```

add_to_down_pool、remove_from_down_pool、add_to_up_pool、remove_from_up_pool：把一个连接加入到下载池/上传池中，功能类似，故只取下载池展示。

```
down_conn_t *add_to_down_pool(down_pool_t *pool, bt_peer_t *peer, chunk_buffer_t
*chunk_buf) {
    down_conn_t *conn = init_down_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;
}

void remove_from_down_pool(down_pool_t *pool, bt_peer_t *peer) {
    down_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;
        }
    }
}
```

5、task_get.c/h

处理来自于用户的调用，主要包括add_and_check_data：用户需要向其他peer请求文件，write_data：收到了包之后写入文件。

```
void add_and_check_data(char *hash, char *data) {
    int i;
    for (i = 0; i < task_get.get_num; i++) {
        char *this_hash = task_get.get_chunks[i].chunk_hash;
        if (memcmp(this_hash, hash, SHA1_HASH_SIZE) == 0) {
            task_get.chunk_data[i] = malloc(BT_CHUNK_SIZE);
            memcpy(task_get.chunk_data[i], data, BT_CHUNK_SIZE);
            break;
        }
    }
    uint8_t hash_my_data[SHA1_HASH_SIZE];
    char *my_data = task_get.chunk_data[i];
    shahash((uint8_t *) my_data, BT_CHUNK_SIZE, hash_my_data);
    task_get.status[i] = memcmp(hash_my_data, hash, SHA1_HASH_SIZE) == 0 ? 1 :
0;
}
```

```
void write_data() {
    FILE *fd = fopen(task_get.output, "wb+");
    for (int i = 0; i < task_get.get_num; i++) {
        fwrite(task_get.chunk_data[i], 1024, 512, fd);
    }
    fclose(fd);
    for (int i = 0; i < task_get.get_num; i++) {
        free(task_get.chunk_data[i]);
    }
    free(task_get.chunk_data);
    free(task_get.status);
    free(task_get.providers);
    free(task_get.get_chunks);
}
```

6、handler.c/h

这一部分的功能比较复杂，主要需要处理WHOHAS分组、IHAVE分组和DATA数据报、ACK分组和丢包，检查本地是否有某个trunk，发送WHOHAS分组、IHAVE分组、DATA数据报、ACK。

当收到其他peer广播的WHOHAS分组时，调用*handle_whohas*：

```
packet_t *handle_whohas(packet_t *pkt_whohas)
{
    int req_num = pkt_whohas->data[0];
    int has_num = 0;
    int from_here = 4;
    char payload[DATALEN];
    char *chunk_hash = pkt_whohas->data + from_here;
    for (int i = 0; i < req_num; i++)
    {
        if (has_chunk(chunk_hash))
        {
            has_num++;
            memcpy(payload + from_here, chunk_hash, SHA1_HASH_SIZE);
            from_here += SHA1_HASH_SIZE;
        }
        chunk_hash += SHA1_HASH_SIZE;
```

```
    }
    if (has_num == 0)
    {
        return NULL;
    }
    else
    {
        memset(payload, 0, 4);
        payload[0] = has_num;
        return new_pkt(IHAVE, HEADERLEN + from_here, 0, 0, payload);
    }
}
```

上述函数需要调用*has_chunk*，主要是检查本地是否有其他peer需要的这个trunk：

```
int has_chunk(char *chunk_hash)
{
    if (chunk_ihave->node_num == 0)
    {
        return 0;
    }
    else
    {
        node_t *curr_node = chunk_ihave->head;
        chunk_t *curr_chunk;
        while (curr_node != NULL)
        {
            curr_chunk = (chunk_t *)(curr_node->data);
          if (memcmp(curr_chunk->chunk_hash, chunk_hash, SHA1_HASH_SIZE) == 0)
            {
                return 1;
            }
            curr_node = curr_node->next;
        }
        return 0;
    }
}
```

如果有这个包的话，构造一个新的package：

```
packet_t *new_pkt(unsigned char type, unsigned short packet_len,
                  unsigned int seq_num, unsigned int ack_num, char *payload)
{
    packet_t *packet = (packet_t *)malloc(sizeof(packet_t));
    packet->header.magic = htons(MAGIC_NUM);
    packet->header.version = 1;
    packet->header.type = type;
    packet->header.header_len = htons(HEADERLEN);
    packet->header.packet_len = htons(packet_len);
    packet->header.seq_num = htonl(seq_num);
    packet->header.ack_num = htonl(ack_num);
    if (payload != NULL)
    {
        memcpy(packet->data, payload, packet_len - HEADERLEN);
    }
    return packet;
```

```
}
```

当peer需要一个trunk时，就向它的所有peer广播WHOHAS分组：

```
list_t *new_whohas_pkt()
{
    list_t *list = init_list();
    char payload[DATALEN];
    unsigned short pkt_len;
    if (task_get.get_num > MAX_CHUNK_NUM)
    { //need more than one packet to send WHOHAS data
        int quotient = task_get.get_num / MAX_CHUNK_NUM;
        int remainder = task_get.get_num - quotient * MAX_CHUNK_NUM;
        for (int i = 0; i < quotient; i++)
        {
            pkt_len = HEADERLEN + 4 + MAX_CHUNK_NUM * SHA1_HASH_SIZE;
            assemble_chunk_hash(payload, MAX_CHUNK_NUM, task_get.get_chunks + i
* MAX_CHUNK_NUM);
            add_node(list, new_pkt(WHOHAS, pkt_len, 0, 0, payload));
        }
        pkt_len = HEADERLEN + 4 + remainder * SHA1_HASH_SIZE;
        assemble_chunk_hash(payload, remainder, task_get.get_chunks + quotient *
MAX_CHUNK_NUM);
        add_node(list, new_pkt(WHOHAS, pkt_len, 0, 0, payload));
    }
    else
    {
        pkt_len = HEADERLEN + 4 + task_get.get_num * SHA1_HASH_SIZE;
        assemble_chunk_hash(payload, task_get.get_num, task_get.get_chunks);
        add_node(list, new_pkt(WHOHAS, pkt_len, 0, 0, payload));
    }
    return list;
}
```

处理IHAVE分组：

```
packet_t *handle_ihave(packet_t *pkt, bt_peer_t *peer)
{
    if (get_down_conn(&down_pool, peer) != NULL)
    {
       return NULL;
    }
    else
    {
        list_t *chunks = split_into_chunks(pkt->data);
        if (down_pool.conn_num >= down_pool.max_conn)
        {
            fprintf(stderr, "download pool is full, please wait!");
            return NULL;
        }
        else
        {
            char *to_download = update_provider(chunks, peer);
            chunk_buffer_t *chunk_buf = init_chunk_buffer(to_download);
            add_to_down_pool(&down_pool, peer, chunk_buf);
```

```
            packet_t *get_pkt = new_pkt(GET, HEADERLEN + SHA1_HASH_SIZE, 0, 0,
to_download);
            return get_pkt;
        }
    }
}
```

当一切准备就绪之后，就可以开始发送分组了，这时需要检查计时器：

```
void handle_get(int sock, packet_t *pkt, bt_peer_t *peer)
{
    up_conn_t *up_conn = get_up_conn(&up_pool, peer);
    //consider the situation of duplicate GET packets
    if (up_conn != NULL)
    {
        remove_from_up_pool(&up_pool, peer);
        up_conn = NULL;
    }
    if (up_pool.conn_num >= up_pool.max_conn)
    {
        fprintf(stderr, "upload pool is full!");
    }
    else
    {
        char to_upload[SHA1_HASH_SIZE];
        memcpy(to_upload, pkt->data, SHA1_HASH_SIZE);
        packet_t **data_pkt = get_data_pkts(to_upload);
        up_conn = add_to_up_pool(&up_pool, peer, data_pkt);
        this_up_conn = up_conn;
        send_data_pkts(up_conn, sock, (struct sockaddr *)(&(peer->addr)));
        alarm(2); //start timer: 2s
    }
}
```

发送和接收package：

```
packet_t **get_data_pkts(char *chunk_hash)
{
    int id;
    for (node_t *node = chunk_tracker->head; node != NULL; node = node->next)
    {
        char *this_chunk_hash = ((chunk_t *)(node->data))->chunk_hash;
        if (memcmp(this_chunk_hash, chunk_hash, SHA1_HASH_SIZE) == 0)
        {
            id = ((chunk_t *)(node->data))->id;
            break;
        }
    }

    FILE *fd = fopen(master_file_name, "r");
    fseek(fd, id * BT_CHUNK_SIZE, SEEK_SET);
    char data[1024];
    packet_t **data_pkts = malloc(CHUNK_SIZE * sizeof(packet_t *));
    for (unsigned int i = 0; i < CHUNK_SIZE; i++)
    {
        fread(data, 1024, 1, fd);
```

```
        data_pkts[i] = new_pkt(DATA, HEADERLEN + 1024, i + 1, 0, data);
    }
    fclose(fd);
    return data_pkts;
}

void send_data_pkts(up_conn_t *conn, int sock, struct sockaddr *to)
{
    int id_sender = config.identity;
    int id_receiver = conn->receiver->id;
    long now_time = clock();
    FILE *fd = fopen("problem2-peer.txt", "at");
    fprintf(fd, "%s%d-%d    %ld    %d\n", "conn", id_sender, id_receiver,
now_time - conn->begin_time, conn->cwnd);
    fclose(fd);
    while (conn->to_send < conn->available)
    {
        spiffy_sendto(sock, conn->pkts[conn->to_send],
                      conn->pkts[conn->to_send]->header.packet_len, 0, to,
sizeof(*to));
        conn->to_send++;
    }
}
```

处理ACK:

```
void handle_ack(int sock, packet_t *pkt, bt_peer_t *peer)
{
    int ack_num = pkt->header.ack_num;

    //to show the change of cwnd more clearly
    FILE *fd = fopen("problem2-peer.txt", "at");
    fprintf(fd, "receive ACK %d\n", ack_num);
    fclose(fd);

    up_conn_t *up_conn = get_up_conn(&up_pool, peer);
    if (up_conn == NULL)
    {
        return NULL;
    }

    if (ack_num == CHUNK_SIZE)
    {
        alarm(0); //stop timer
        remove_from_up_pool(&up_pool, peer);
    }
    else if (ack_num > up_conn->last_ack)
    {
        alarm(0);
        up_conn->last_ack = ack_num;
        up_conn->dup_times = 0;
        //up_conn->to_send = up_conn->to_send > ack_num ? up_conn->to_send :
ack_num;
        int next_available = ack_num + up_conn->cwnd;
        up_conn->available = next_available <= CHUNK_SIZE ? next_available :
CHUNK_SIZE;
```

```
        send_data_pkts(up_conn, sock, (struct sockaddr *)(&(peer->addr)));
        alarm(2);
    }
    else if (ack_num == up_conn->last_ack)
    {
        up_conn->dup_times++;
        if (up_conn->dup_times >= 3)
        {
            alarm(0);
            up_conn->dup_times = 0;
            up_conn->to_send = ack_num;
            int available = up_conn->to_send + up_conn->cwnd;
           up_conn->available = available < CHUNK_SIZE ? available : CHUNK_SIZE;
            send_data_pkts(up_conn, sock, (struct sockaddr *)(&(peer->addr)));
            alarm(2);
        }
    }
}
```

处理超时，如果发生超时需要重发并且重设计时器和窗口：

```
void handle_timeout()
{
    this_up_conn->to_send = this_up_conn->last_ack;
    this_up_conn->dup_times = 0;
    int next_available = this_up_conn->to_send + this_up_conn->cwnd;
    this_up_conn->available = next_available <= CHUNK_SIZE ? next_available :
CHUNK_SIZE;
    send_data_pkts(this_up_conn, config.sock, (struct sockaddr *)(&
(this_up_conn->receiver->addr)));
    alarm(2);
}
```

main callers

到此为止我已经介绍了用到的数据结构和辅助函数，实际上一个类似于tcp的连接已经被完成了（除了没有拥塞控制部分），现在只需要在此基础上调用它们，构建一个p2p应用，这主要是在peer.c中完成。

在peer.c的main函数中，首先也是进行一些必要的准备工作例如初始化chunks_ihave、down_pool和up_pool，接下来进行到函数的主要工作*peer_run*。

peer_run构建好用于通信的socket之后，便进入到一个while(1)的循环中，主要是处理两类事件：来自用户的输入和peers间的信号或包的到达。

调用函数*process_inbound_udp*来处理peers之间的信号或包的到达，主要有WHOHAS/IHAVE/GET/DATA/ACK/DENIED，根据包的头部的这些标志位的不同调用不同的handler函数：

```
void process_inbound_udp(int sock) {
#define BUFLEN 1500
    struct sockaddr_in from;
    socklen_t fromlen;
    char buf[BUFLEN];
    fromlen = sizeof(from);
    spiffy_recvfrom(sock, buf, BUFLEN, 0, (struct sockaddr *) &from, &fromlen);
```

```
//    printf("PROCESS_INBOUND_UDP SKELETON -- replace!\n"
//           "Incoming message from %s:%d\n%s\n\n",
//           inet_ntoa(from.sin_addr),
//           ntohs(from.sin_port),
//           buf);
    packet_t *pkt_rcv = (packet_t *) buf;
    pkt_ntoh(pkt_rcv);
    int pkt_type = parse_type(pkt_rcv);
    //find the peer who sends the packet
    bt_peer_t *peer;
    for (peer = config.peers; peer != NULL; peer = peer->next) {
        if (peer->addr.sin_port == from.sin_port) {
            break;
        }
    }

    switch (pkt_type) {
        case WHOHAS: {
            packet_t *reply = handle_whohas(pkt_rcv);
            if (reply != NULL) {
                spiffy_send_to(sock, reply, reply->header.packet_len, 0, (struct
sockaddr *) &from, fromlen);
                free(reply);
            }
            break;
        }
        case IHAVE: {
            packet_t *get_pkt = handle_ihave(pkt_rcv, peer);
            if (get_pkt != NULL) {
                spiffy_sendto(sock, get_pkt, get_pkt->header.packet_len, 0,
(struct sockaddr *) &from, fromlen);
                free(get_pkt);
            }
            break;
        }
        case GET: {
            handle_get(sock, pkt_rcv, peer);
            break;
        }
        case DATA: {
            handle_data(sock, pkt_rcv, peer);
            break;
        }
        case ACK: {
            handle_ack(sock, pkt_rcv, peer);
            break;
        }
        case DENIED:
            //do nothing
            break;
        default:
            printf("Invalid Packet Type: %d\n", pkt_type);
    }
}
```

另一方面调用*process_user_input*处理用户输入，而*process_user_input*进一步调用*process_get*来构造WHOHAS包、发起一个新的连接并完成一个下载的完整流程：

```
void process_get(char *chunkfile, char *outputfile) {
//    printf("PROCESS GET SKELETON CODE CALLED.  Fill me in!  (%s, %s)\n",
//           chunkfile, outputfile);

    init_task(chunkfile, outputfile);
    list_t *ask_whohas = new_whohas_pkt();
    node_t *node = ask_whohas->head;
    while (node != NULL) {
        packet_t *packet = (packet_t *) (node->data);
        bt_peer_t *curr_peer = config.peers;
        while (curr_peer != NULL) {
            if (curr_peer->id != config.identity) {
                struct sockaddr *to = (struct sockaddr *) &(curr_peer->addr);
                spiffy_sendto(config.sock, packet, packet->header.packet_len, 0,
to, sizeof(*to));
            }
            curr_peer = curr_peer->next;
        }
        free(packet);
        node = node->next;
    }
}
```