计算机网络PJ设计与说明文档

宣子涛 18302010015

一、Auto Testing通过情况概述

1. checkpoint1　　通过✅
2. checkpoint2　　通过✅
3. checkpoint3　　通过✅
4. concurrenttest　通过✅

二、基本架构设计与思想

2.1 设计思想

本PJ设计秉承着精简设计的原则，在能够实现基本数据传输以及传输的可靠性、并发性和鲁棒性的基础上，尽可能采取最简洁有效的设计。尤其是在重传方案的选取上，本设计摒弃了耗费大量CPU资源的多线程方案和繁琐的单线程多alarm/SIGALRM方案，而采取了I/O多路复用+select超时方案，达到了用精简的代码实现高性能的效果（三种重传方案的介绍和取舍详见4.1.1节）。

2.2 函数设计概述

除了peer.c等预先提供的代码架构和文件外，本设计添加了packet.h、handler.c(.h)和utils.c(.h)。

handler.c(.h)中提供了一系列用于处理stdin或其他peer发送的报文的函数，如下所示：

```
void deal_with_ACK(data_packet_t *curr, struct sockaddr_in *from);
void deal_with_DATA(data_packet_t *curr, struct sockaddr_in *from);
void deal_with_GET(data_packet_t *curr, struct sockaddr_in *from);
void deal_with_IHAVE(data_packet_t *curr, struct sockaddr_in *from);
void deal_with_WHOHAS(data_packet_t *curr, struct sockaddr_in *from);
void send_packet(data_packet_t packet, struct sockaddr_in *from);
```

为了实现代码结构的精简和有效复用，在utils.c(.h)中提供了一系列抽取出的可复用方法，如下所示：

```
header_t make_packet_header(char packet_type, short header_len, short
packet_len, u_int seq_num, u_int ack_num);
int check_hash(unsigned char *data, unsigned char *hash);
int search_id_for_hash(char *request_hash_name, char *file_name);
void search_master_tar(char *file_name);
int find_free_upload_info();
int find_free_waiting_entity();
int find_free_download_info();
void try_send_GET_in_waiting_list();
```

2.3 结构体设计概述

packet.h中基于预先提供的server.c和client.c中的报文设计方案，定义了本设计的报文结构，如下所示：

```
// 报文的头部结构
typedef struct header_s
{
    // magic number，为定值15441
    short magicnum;
    // version，为定值0x01
    char version;
    // 根据报文的种类不同，有不同的packet type，取值从0x00到0x05
    char packet_type;
    // 报文头部的长度
    short header_len;
    // 报文的总长度
    short packet_len;
    // 发送的DATA序号，仅在DATA类型的报文中存在
    u_int seq_num;
    // 发送的ACK号，仅在ACK类型的报文中存在
    u_int ack_num;
} header_t;
// 报文的整体结构
typedef struct data_packet
{
    header_t header;
    // 用于存放Chunk Hash或文件DATA
    unsigned char data[2000];
} data_packet_t;
```

此外，为了实现传输的可靠性、并发性和鲁棒性，保存传输的一些有效信息，我还设计了几个额外的结构体。

首先是upload_to_peer_info，记录了本peer作为“服务器”（发送方）时，其他向我方请求数据的peer信息。

```
typedef struct upload_to_peer_info
{
    // 其他peer所请求的Chunk Hash在Master Chunks中的id
    int request_id;
    // Master Chunk的文件名
    char master_tar_name[20];
    // 向我方请求资源的peer的addr信息
    struct sockaddr_in sockaddr_upload_to;
    // 最后一次向该peer发送DATA的时间，用于判断超时
    struct timeval last_sent_time;
    // 当前收到的最大的ack
    int max_ack;
    int status;
} upload_to_peer_info_t;
```

与之对应的download_from_peer_info记录了本peer作为“客户端”（请求方）时，正在向对方请求数据的peer信息。

```
typedef struct download_from_peer_info
{
    // 当前请求的Chunk Hash在Request Chunks中的id
    int request_id;
    // 向对方请求数据的peer信息
    struct sockaddr_in sockaddr_download_from;
    // 当前待写入的位置（在当前请求的chunk中的偏移值，即在定长512KB的chunk中的偏移值，而不是在整个文件中的偏移值）
    int curr_pos;
    // 期望等待的DATA的序号
    int expected_seq;
    int status;
} download_from_peer_info_t;
```

然后是chunk_status，记录了本peer作为“客户端”（请求方）时，要请求的文件的每个chunk的信息及状态（未传输、正在传输中和已传输完成并经过校验）。

```
typedef struct chunk_status
{
    int status;
    // 在被请求文件中的id（而不是在master chunks中的id）
    int id;
    // 在被请求文件中的id对应的hash
    unsigned char hash[LEN_OF_CHUNK_HASH];
} chunk_status_t;
```

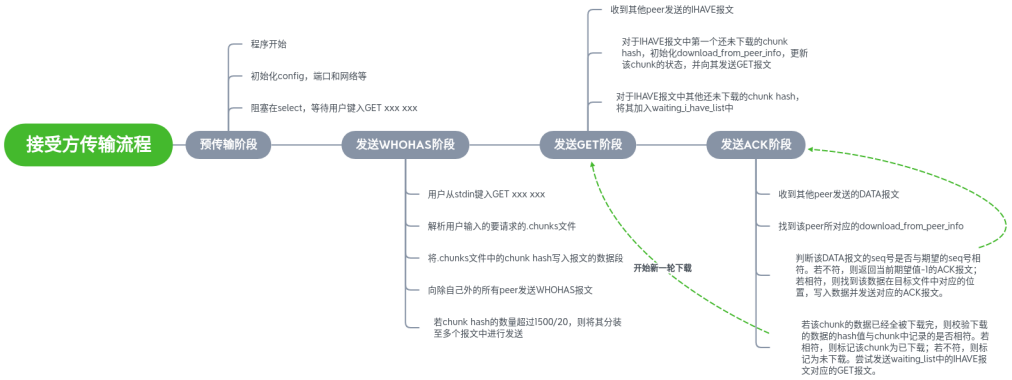
最后是waiting_I_HAVE。由于每两个peer之间同时最多只能建立一个连接，所以当一个peer向另一个peer发送的IHAVE报文中存放了超过一个chunk hash时，可以在这个结构体中存放已经收到IHAVE但还没发送GET的chunk hash及其对应的peer信息。这样可以在一定程度上缓解reflood WHOHAS，而且是符合设计逻辑的（不然每次IHAVE在绝大多数情况下只需发送一个chunk hash即可，而不必发送对应与WHOHAS的所有自己拥有的chunk hash）。在经过一定时间间隔后，程序会调用前文所述utils.c中的try_send_GET_in_waiting_list()函数，尝试发送/清理waiting_I_HAVE_list。结构体设计如下所示：

```
// 记录了某个peer节点所拥有的对应于要请求的Chunk，但由于当前正在从该peer下载数据，而必须等待并在之后再发GET请求进行下载（如有需要的话）
typedef struct waiting_I_HAVE
{
    // 拥有该chunk的addr
    struct sockaddr_in sockaddr_download_from;
    // chunk的hash
    unsigned char hash_name[20];
    // 当前请求的Chunk Hash在Request Chunks中的id
    int request_id;
    int status;
} waiting_i_have_t;
```

三、基本传输流程说明

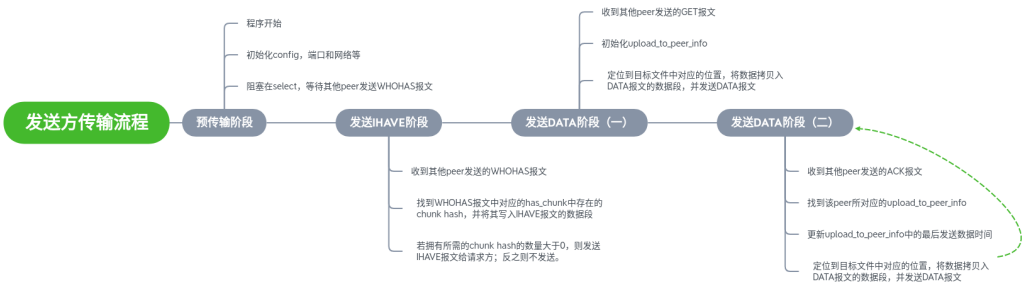
首先介绍接受方和发送方各自基本的传输流程，可靠性（重传）和并发性等实现会在第4节中详细介绍。

3.1 接受方/请求方



如图所示，接受方在程序启动后，会在初始化后阻塞在select（此时还不一定是接受方，下同）。当用户在stdin键入GET XXX YYY后，正式成为接受方。根据用户键入的第一个参数读取要请求的chunks，并将所有hash写入WHOHAS报文，向除自己外的所有peer发送这一WHOHAS报文。如果要请求的chunk的数量大于1500（报文数据段最大长度）/20（hash的固定长度），则需要将其分装至多个报文中进行发送。而后收到其他peer发送的IHAVE报文后，对于IHAVE报文中第一个还未下载的chunk hash，初始化download_from_peer_info，更新该chunk的状态，并向其发送GET报文；对于IHAVE报文中其他还未下载的chunk hash，将其加入waiting_i_have_list中。接着，对于这一连接，收到该peer发送的DATA报文，判断该DATA报文的seq号是否与期望的seq号相符。若不符，则返回当前期望值-1的ACK报文；若相符，则找到该数据在目标文件中对应的位置，写入数据并发送对应的ACK报文。若该chunk的数据已经全被下载完，则校验下载的数据的hash值与chunk中记录的是否相符。若相符，则标记该chunk为已下载；若不符，则标记为未下载。尝试发送waiting_list中的IHAVE报文对应的GET报文。

3.2 发送方



如图所示，发送方在程序启动后，会在初始化后阻塞在select。收到其他peer发送的WHOHAS报文后，找到WHOHAS报文中对应的has_chunk中存在的chunk hash，并将其写入IHAVE报文的数据段。若拥有所需的chunk hash的数量大于0，则发送IHAVE报文给请求方；反之则不发送。接着，收到peer发送的GET报文，初始化upload_to_peer_info，定位到目标文件中对应的位置，将数据拷贝入DATA报文的数据段，并发送DATA报文。对于这一连接而言，发送方再次收到对应peer发送的ACK，更新upload_to_peer_info中的最后发送数据时间，定位到目标文件中对应的位置，将数据拷贝入DATA报文的数据段，并发送DATA报文。

四、传输的可靠性、并发性和鲁棒性设计

4.1 可靠性

对于网络传输的可靠性而言，其最大也是最现实的威胁便是丢包。在网络传输中，无法保证一个数据报是否能够成功地传输给接受方。这样一来，为了实现网络传输的可靠性，保证即使有丢包的情况发生，也能够在最终准确无误地把数据发送给接受方，一个有效的重传机制是必不可少的。一个有效且高性能的重传机制对于一个网络传输应用而言是至关重要的。

如前（2.1节）所述，有三种重传的实现方式，分别是：多线程方案、单线程多alarm/SIGALRM方案和本设计中所使用的I/O多路复用+select超时方案。接下来，我会对这三种方案进行详细的描述，并阐释为什么本设计中采用了I/O多路复用+select超时方案。

4.1.1 重传的三种方案详述

首先介绍**多线程方案**。

判断超时的最简单的也是最直观的一种方法便是，sleep指定一段时间（超时时间），若在sleep了超时时间后还未收到另一方发送的ack相应，则猜测发生了丢包事件，启动重传机制。然而，这一方法最大的问题是，这是一种阻塞的方法。在sleep期间，无法接受/处理其他任何peer发送的任何报文，必须等到sleep结束后才能继续处理，这样会使得传输的效率大幅下降，尤其是对于并发的情况下，原本可以利用网络传输的时间提高传输效率（比如，一个peer在等待另一个peer发送数据的时候，可以处理其他peer已经来到的数据），如果直接使用sleep则会使效率与串行的情况的效率相近。为了处理这一问题，也有一个很直观（但代码量却很大）的解决方案，也就是我所说的多线程方案：为每一个连接开辟一个新的线程，每个发送方线程在发送完DATA报文后sleep一段时间，若在sleep结束后，还未收到接受方的ACK报文，则启动重传机制。

这样一来，实现了重传，且可以同时处理多个peer的请求，也使并发变成了真正的并行，相比使用sleep实现的串行化的效率会有提升，但其仍有许多问题。

首要的问题是，sleep是一种阻塞固定时间的解决方案，即使报文在sleep结束之前到来，也无法立刻开始处理报文。这就会导致每发送完一个报文后，都需要经过固定的超时时间才能发送下一个报文，仍然会导致效率低下。对应的解决方案是使用recvfrom替代sleep，并将recvfrom的超时时间设置为sleep的时间。这样一来，如果ack报文在超时之前提前到来，便可以立刻开始处理，发送下一个DATA报文，大大提高传输效率。

然而，recvfrom作为一种阻塞的解决方案，其同时也只能处理一个与其他peer的连接，仍然需要使用多线程。而多线程会带来CPU占用率的提升，而且每个线程只处理一个连接，是与提供的peer.c框架中所使用的select I/O多路复用是矛盾的（因为每个线程只处理一个连接，根本用不到I/O多路复用）。

其次是**alarm/SIGALRM方案**。相比于sleep/recvfrom的多线程阻塞方案，alarm/SIGALRM提供了一种非阻塞的方式。发送方可以在发完DATA报文后，使用alarm函数设定超时时间，并用signal函数将SIGALRM的处理函数重定义为重传函数。在使用alarm函数设定超时时间后，程序能非阻塞地继续运行，处理来自其他peer的其他报文，直到alarm超时，调用SIGALRM的处理函数执行超时重传。

这是一种看似美好且逻辑顺畅的解决方案，然而它却有一个致命的缺陷：一个线程/进程只能对应一个alarm，如果在前一个alarm未超时时设定新的alarm，新的alarm时间会覆盖旧的alarm时间，而旧的alarm时间会被丢弃。考虑这样一种情况，peer1向peer2发送了一个DATA报文并设置了alarm，而后收到peer3的请求并向peer3发送DATA报文并设置新的alarm，这时前一个alarm还没超时，而新的alarm会覆盖旧的alarm，假设peer1最初向peer2发送的DATA报文丢失，那么就无法启动重传机制而一直阻塞。当然也有解决方案，一是使用额外的数据结构为每一个连接设置一个alarm（可以在单线程中实现，大致为设定一个基准时间的alarm，每

个连接在这个基准的alarm的基础上设置超时时间，不详述），然而这显然是违反操作系统/硬件逻辑且会带来巨大的性能损耗的（否则操作系统/C语言为什么不直接为每个线程提供多个alarm）；二是每当alarm超时时，向所有正在连接中的节点进行重传（然而，这一方案会增加网络的占用，且仍有潜在的永远阻塞的可能：如果在丢失的alarm的丢包后再也无超时，那就无法启动重传）；三是使用多线程，每个线程启动一个alarm，其类似于前文的多线程方案，其缺点也相同。

最后是**I/O多路复用+select超时方案**，也是我最终使用的方案。首先在发送方的每一个连接的信息（upload_to_peer_info结构体）中记录了最后发送时间（last_sent_time），每当发送方向连接的另一方发送数据后，更新此连接的最后发送时间为当前时间。而后，每当发送方发完一份DATA报文后，遍历所有当前连接的信息（upload_to_peer_infos），找到其中离当前时间最远的最后发送时间，而后将select的阻塞时间设置为：(超时时间-(当前时间-最远的最后发送时间))。当select超时时，同样遍历所有当前连接的信息，找到超时的连接并执行重传。这一方案利用了select所提供的超时机制，与peer.c提供的框架逻辑相符合，具有较好的性能，不会出现永远阻塞的情况（因为每次发送DATA报文后便会更新select的阻塞时间，如果设select的阻塞时间为定值的话，则可能出现永远阻塞的情况），而且代码实现简单，只需为连接的信息（upload_to_peer_info结构体）中添加一个属性即可。

4.1.2 重传的三种方案之间的比较与取舍

前文在描述三种方案时已经叙述了其优缺点，在此做一综述。

一、使用**多线程方案**会带来CPU占用率的提升，而且每个线程只处理一个连接，是与提供的peer.c框架中所使用的select I/O多路复用是矛盾的，且代码实现复杂；

二、使用**alarm/SIGALRM方案**有潜在的永远阻塞的可能或违反操作系统/硬件逻辑并带来巨大的性能损耗；

三、使用**I/O多路复用+select超时方案**与peer.c提供的框架逻辑相符合，具有较好的性能，不会出现永远阻塞的情况，且代码实现简单。

因此，我最终采用了第三种方案。

4.2 并发性

预先提供的peer.c中的框架中所使用的select I/O多路复用为并发性的实现提供了良好的基础。使用I/O多路复用的事件驱动编程，不需要使用多线程（实际是并行）就能实现并发。对于网络传输这样的传输过程需要较长且不确定时间的应用而言，使用并发就可以充分利用这一段等待传输的时间，大幅提高传输效率。

就具体的实现而言，我使用了一个数组（upload_to_peer_infos与download_from_peer_infos）来记录当前正在传输中的连接。这一数组中存放着连接的信息，每当一个报文来到时，便遍历这一数组，根据这一报文的发送方的地址找到对应的连接信息（如expect_seq，request_id等），从而再根据连接信息，执行相应的操作（比如打开对应的文件并seek到对应的位置，开始读或写数据）。此外，每一个连接还有一个状态信息，用于记录该连接是否正在忙碌中。当连接初始化时，从数组中找到一个空闲的连接位置，并将其状态值设为忙碌中；当结束发送/接收后，将这一连接memset为0，供后续使用。如果使用一个链表来记录这些连接信息的话，就不需要这一状态信息了。这样一来，发送方和接受方就能够记录正在进行传输中的连接，并找到对应的信息，从而可以同时开启与不同的peer的多条连接，实现并发性。

此外，为了不重复向多个peer请求同一chunk，我使用了chunk_status数组记录了要请求的每个chunk的信息和当前状态，可以分为待下载、下载中和已下载。

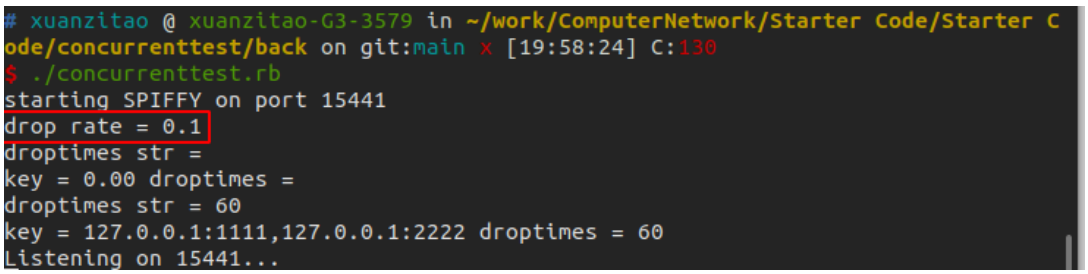
以concurrenttest为例介绍具体流程。peer3会向peer1和peer2发送WHOHAS，且peer1和peer2都有peer3所需要的所有数据。于是，peer3收到peer1的IHAVE后，向peer1发送GET，请求所需的第一个chunk，并在download_from_peer_infos数组中找到一个状态为空闲的位置（download_from_peer_infos[0]），将其与peer1的连接信息记录进这一位置（如

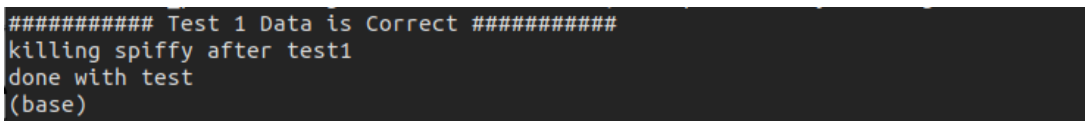
request_id等信息）并将第一个chunk的状态置为下载中。再之后peer3收到peer2的IHAVE，peer3遍历chunk_status数组，发现第一个chunk已经正在下载中，而第二个chunk待下载，于是向peer2发GET，请求所需的第二个chunk，并在download_from_peer_infos数组中找到一个状态为空闲的位置（download_from_peer_infos[1]），将其与peer2的连接信息记录进这一位置（如request_id等信息）并将第二个chunk的状态置为下载中。如此一来，peer3便可以同时开启与peer1和peer2的连接，实现并发性。

4.3 鲁棒性

4.3.1 同时应对丢包+并发的情况

在check point中，cp3测试了能否应对丢包的情况，concurrenttest测试了能否应对并发的状况，而没有测试同时应对丢包+并发的情况。我将concurrenttest中hupsim_adv.pl文件中的rand_drop_rate改为0.1，并测试此情况（同时有丢包和并发）下系统是否仍能正常工作。经测试，这一情况下系统仍能正常通过测试，从而证明了系统具有较好的鲁棒性。结果如图所示，视频中也有相应的展示。





4.3.2 应对corrupt数据的情况

在接受方接受完一个chunk的所有数据后，会使用check_hash函数（如下所示）判断接收的数据的hash值和chunk file中记录的hash是否一致。如果一致，再将chunk_status中对应chunk的状态改为完成下载；如果不一致，则清空数据并将chunk_status中对应chunk的状态改为待下载。

```
int check_hash(unsigned char *data, unsigned char *hash)
{
    unsigned char data_hash[LEN_OF_CHUNK_HASH];
    shahash(data, BT_CHUNK_SIZE, data_hash);
    return memcmp(data_hash, hash, LEN_OF_CHUNK_HASH);
}
```