Distributed P2P Communication Software(DP2PCS)

https://github.com/ukari/DP2PCSC

write by Muromi Ukari

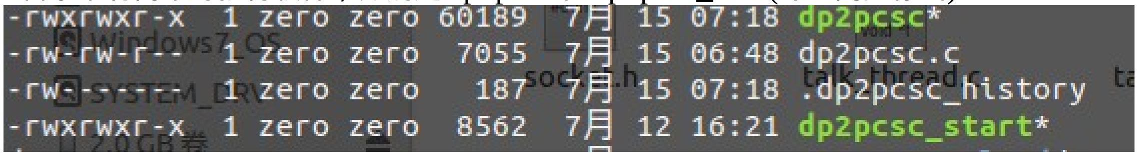
一.简单介绍

1.这是一个通讯软件
2.可以多对多进行聊天
3.可以多对多收发各类型文件
4.可以方便的添加好友
5.这是一个分布式 P2P 软件，可以在不需要任何服务器的情况下运行
6.好友信息需要在文本文件中配置
7.该软件为好友管理服务器留下接口

二.功能描述

以下的功能描述以一次完整的内网通讯过程为例，通讯的双方分别为 ukari 和 jhon titor。
ukari 的 ip 地址是 192.168.0.1
jhon titor 的 ip 地址是 192.168.0.100
1.运行软件

目录下有两个可执行软件，分别是 dp2pcsc 和 dp2pcsc_start(绿色高亮标出)



```
-rwxrwxr-x  1 zero zero 60189  7月 15 07:18 dp2pcsc*
-rw-rw-r--  1 zero zero  7055  7月 15 06:48 dp2pcsc.c
-rw-------  1 zero zero   187  7月 15 07:18 .dp2pcsc_history
-rwxrwxr-x  1 zero zero  8562  7月 12 16:21 dp2pcsc_start*
```

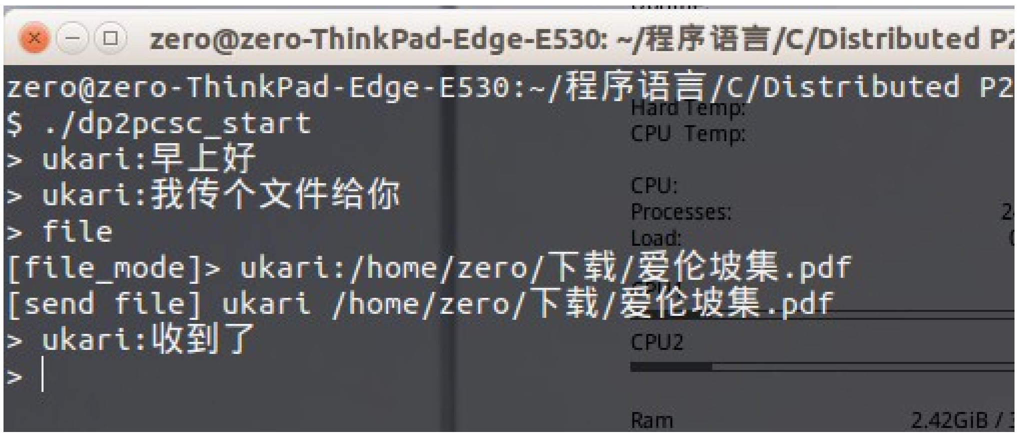
运行 dp2pcsc_start 需要预装 rlwrap 软件。Ubuntu 系统下，联网输入 sudo apt-get install rlwrap 即可运行。

dp2pcsc 与 dp2pcsc_start 的基本功能完全一样，区别在行编辑的功能上。

dp2pcsc_start 可以更加美观的输入中文，同时光标可以在已输入文本中前后移动，通过上下键甚至可以查看历史命令。

2.软件界面

软件打开后会出现两个窗体，一个用于输入命令，一个用于显示消息。



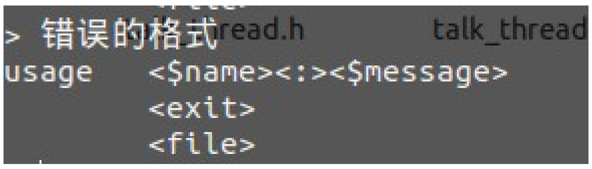
```
zero@zero-ThinkPad-Edge-E530:~/程序语言/C/Distributed P2
$ ./dp2pcsc_start
> ukari:早上好
> ukari:我传个文件给你
> file
[file_mode]> ukari:/home/zero/下载/爱伦坡集.pdf
[send file] ukari /home/zero/下载/爱伦坡集.pdf
> ukari:收到了
> |
```

Jhon Titor 输入命令的窗体

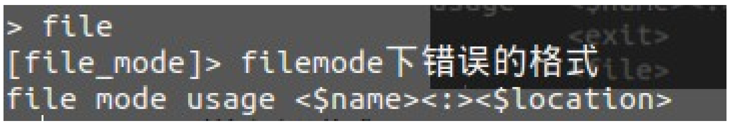
该软件实现了一个简单的解析器。

输入命令有三种，分别为默认，file_mode，exit。

当命令输入格式错误时，软件会给出命令格式的 usage

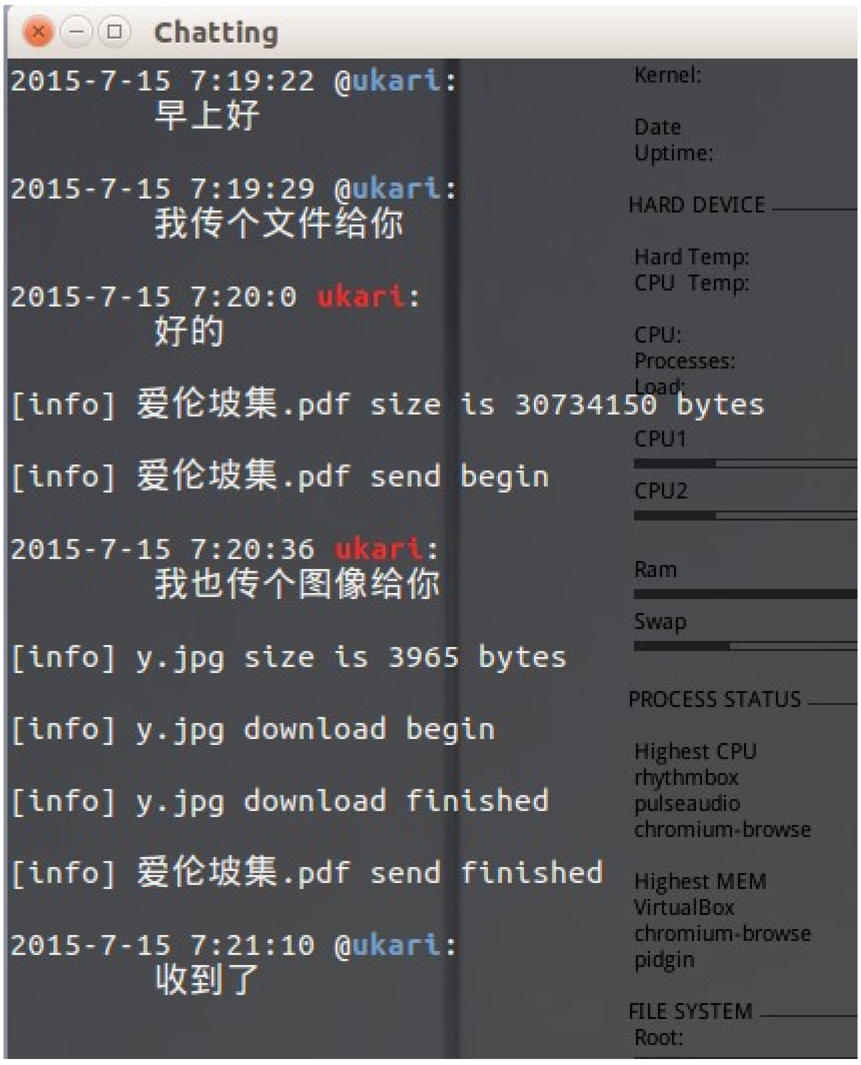


```
> 错误的格式
usage    <$name><:><$message>
         <exit>
         <file>
```


```
> file
[file_mode]> filemode下错误的格式
file mode usage <$name><:><$location>
```

显示消息的窗体(名为 Chatting)中，有三种输出格式。
接收到好友的消息时，好友名字会红色高亮，同时显示消息的到达时间。
发送给好友消息时，好友名字会用@+蓝色高亮进行标识，同时显示消息的到达时间。
Info 格式，普通的显示系统的消息，如文件下载/发送完毕。

此外，当 Chatting 被关闭的时候，DP2PCSC 的 Chatting 守护进程将自动重新运行 Chatting，Chatting 将在 DP2PCSC 通过 exit 命令退出时自动关闭。



```
2015-7-15 7:19:22 @ukari:
        早上好

2015-7-15 7:19:29 @ukari:
        我传个文件给你

2015-7-15 7:20:0 ukari:
        好的

[info] 爱伦坡集.pdf size is 30734150 bytes

[info] 爱伦坡集.pdf send begin

2015-7-15 7:20:36 ukari:
        我也传个图像给你

[info] y.jpg size is 3965 bytes

[info] y.jpg download begin

[info] y.jpg download finished

[info] 爱伦坡集.pdf send finished

2015-7-15 7:21:10 @ukari:
        收到了
```

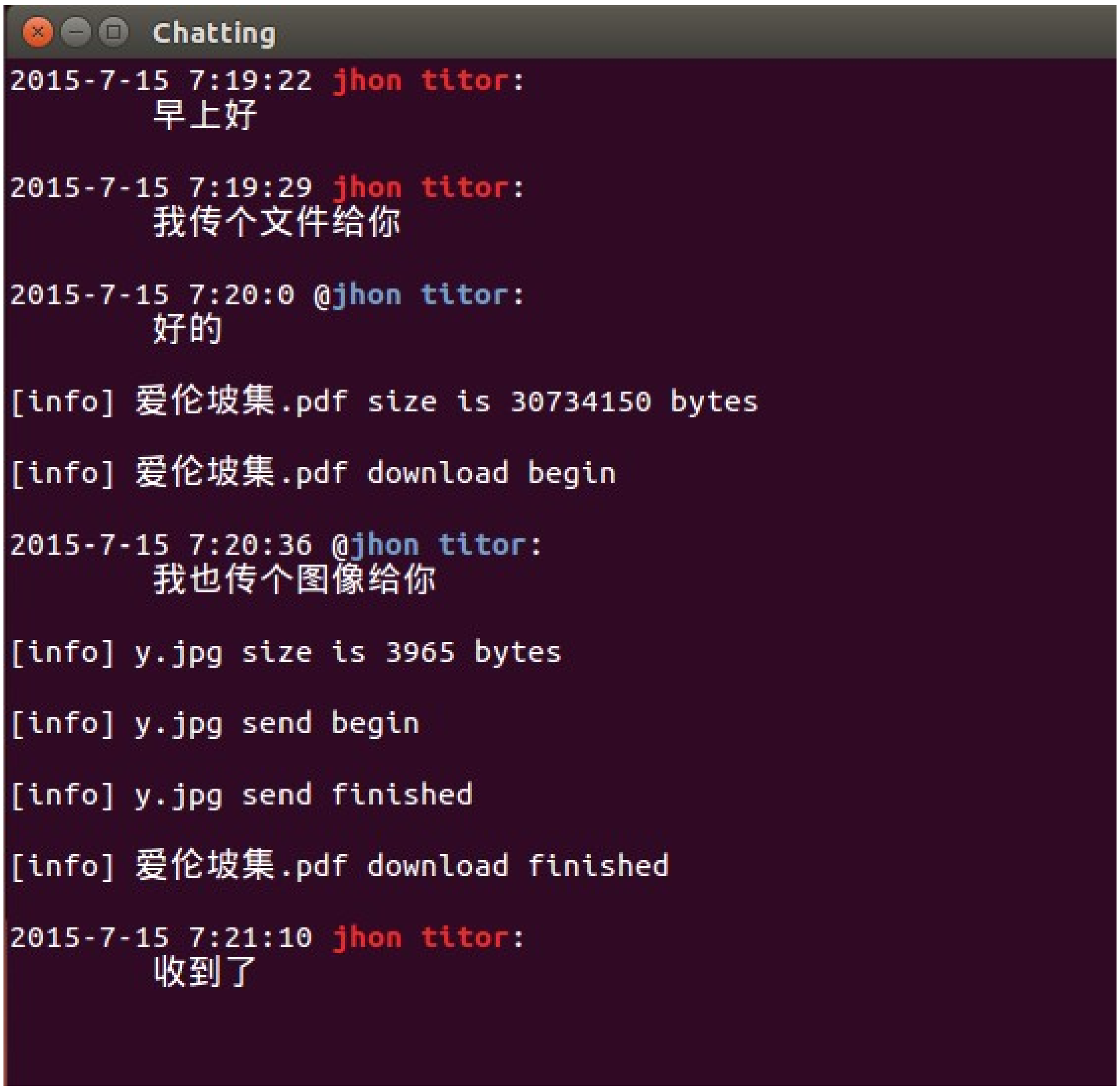
Jhon Titor 的显示消息的窗体

3.聊天功能

默认输入模式下，输入"好友名:待发送消息"，如果好友在好友列表中存在的话，系统将查询好友的 Ip，然后向对方发送消息。如果是本次登陆中第一次与该好友通讯，那么一条专用的数据通道将建立起来，之后的消息通讯都会在这条数据通道上进行。如果好友离线了，与该好友之前建立的数据通道将断开。

与每个好友各自只能建立至多一条数据链路用于消息通讯。

任何好友发来的消息和用户发出到好友的消息都会显示在 Chatting 窗体中，发送者和发送方向会被很好的区分标识出来。



```
2015-7-15 7:19:22 jhon titor:
        早上好

2015-7-15 7:19:29 jhon titor:
        我传个文件给你

2015-7-15 7:20:0 @jhon titor:
        好的

[info] 爱伦坡集.pdf size is 30734150 bytes

[info] 爱伦坡集.pdf download begin

2015-7-15 7:20:36 @jhon titor:
        我也传个图像给你

[info] y.jpg size is 3965 bytes

[info] y.jpg send begin

[info] y.jpg send finished

[info] 爱伦坡集.pdf download finished

2015-7-15 7:21:10 jhon titor:
        收到了
```

Ukari 显示消息的 Chatting 窗体

4.文件传输功能

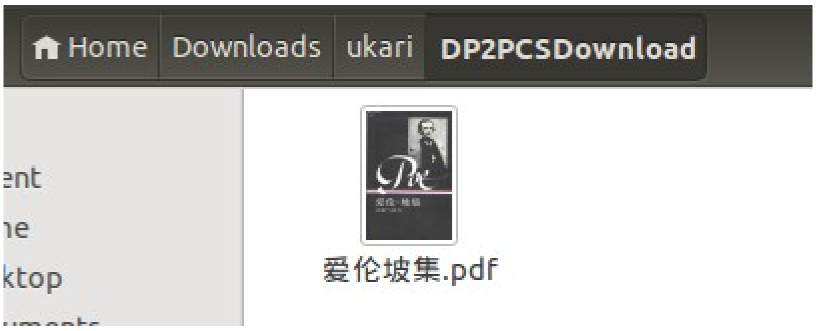
默认输入模式下，输入 file 可以进入 file_mode 模式中，在 file_mode 模式中，输入"要发送的好友名:要发送的文件的路径"，文件将被发送给对方。文件路径可以是绝对路径或是相对路径。

文件传输开始时，Chatting 窗口将给出待传输文件大小和待传输文件名的提示。

文件传输在后台新建的专用数据通道进行，每个文件都会独自建立专用的数据通道，多个文件可以同时接收和发送。

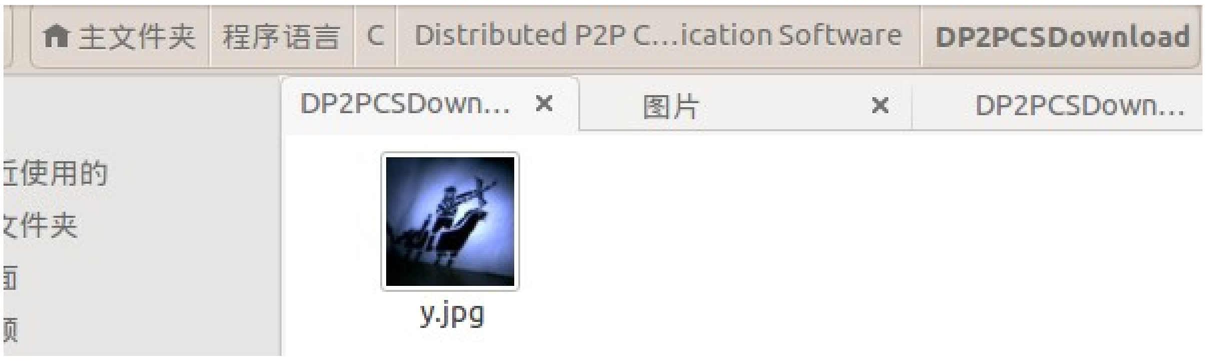
当文件传输结束时，该文件打开的专用数据通道将关闭。

如果文件传输错误发生，任何的传送方主动或意外的关闭 DP2PCSC，对方的文件数据通道也将自动关闭。



Jhon Titor 传给 Ukari 的 pdf 文件

Jhon Titor 传给 Ukari 的 pdf 文件



Ukari 传给 Jhon Titor 的图片文件

三.设计思路

通讯和社交是组成人们精彩生活中必不可少的部分，而齐默尔曼定律[1]说道——"技术自然而然的倾向于使监控变得更加容易进行的方向发展，并且计算机系统追踪人们的能力每 18 个月翻一倍"，由于这个原因，在未来的社会生活中，健壮、分布式、难以被监听的通讯软件将会获得越来越广泛的市场。
因此，我准备制作一套分布式点对点的通讯软件，即 Distributed P2P Communication Software(DP2PCS)，由 Distributed P2P Communication Software Client(DP2PCSC)和 Distributed P2P Communication Software Server(DP2PCSS)两部分组成。
DP2PCSC 由普通用户运行，可以通过类似静态 IP 配置或动态 IP 解析的技术获取自己联系好友的 Ip 地址，并建立直接连接。
DP2PCSS 由服务供应者运行，其功能是维护一组{用户名=>用户登陆 IP，用户名=>好友名}的映射表，管理好友信息，和按照 DP2PCSC 端的请求发给 DP2PCSC 端好友 Ip 地址表，或者传送单个表项，便于

Client 端个别更新。通讯数据不会经过 DP2PCSS。
本次作业中，DP2PCSC 采用静态 IP 配置的方法，即可满足题目需求。

DP2PCSC 的架构设计

功能模块层：监听模块 文件传输模块 消息传输模块 ———— 显示模块 ———— 命令输入模块
|
系统控制层(队列管理器)
|
数据结构层 ———— 基本库 ———— 配置文件
|
数据库层

数据库层: friend_address 的 name 到 ip address 映射的纯文本文件
数据结构层: 通用队列 Queue.c，基本库 ulib.c，配置文件 socket.h
系统控制层(队列管理器): 好友名到地址映射队列 friend.c， 建立连接的数据通路队列 friend.c，文件传输控制队列 file_trans.c 该层的所有读写操作都有加对应的读写锁。
监听模块: listen_thread.c
文件传输模块: trans_file.c talk_thread.c
消息传输模块: talk_thread.c
命令输入模块:dp2pcsc.c
显示模块:show_thread.c show_tty_daemon.sh

基本做到了上层模块调用下层模块，下层模块不知道上层模块。
其中，我最满意的是系统调用层的 file_trans 队列控制器的设计，采用的注册，注销的模式，使用效果和 socket 函数一样返回一个 int 值，这样通过这个 int 返回值就能够对打开的 file_trans 进行操作。

通讯协议设计

In talk_thread.c		
Is connect launcher	Launcher	Receiver
Set connect_type	//Check arg set connect_type	Recv head control data (connect_type)
	//Send connect_type + ETB	
	//Recv ACK + ETB	Set connect_type
		Send ACK + ETB
Message connect	while	while

	Recv and show	Recv and show
	done	done
File connect	Launcher is Putter	Accept is Downloader
Trans file name	Send filename from arg	Recv filename + ETB
Will file send request be accept by Downloader?	Recv CAN/ACK	Wait for main thread <fileaccept> <filerefused>
		Send ACK in <fileaccept> /Send CAN in <filefused>
	If recv ACK continue If recv CAN close socket and exit thread	If send ACK continue If send CAN close socket and exit thread
Trans file blocks	While file end	While EOT recv
	Send file block	Recv file block + ETB
	Recv ACK	Send ACK + ETB
	done	done
	Send EOT + ETB	destroy
	destroy	

该协议在实际的实践中出现了问题，对于文件传输模块，ETB 可能会和要发送的文件数据本身重复，造成数据块的区分失败，虽然可以通过 base64 将要发送的文件数据编码传送，但是这样会造成传输效率接近 33%的下降，算上 base64 的编码和解码时间，传输的效率将进一步降低。因此后期传送文件时通过文件总长度和已经传输长度的比较控制传输，取得的很好的结果。

环境和工具

环境: 3.13.0-57-generic #95-Ubuntu SMP Fri Jun 19 09:28:15 UTC 2015 x86_64 x86_64 x86_64 GNU/Linux

项目管理: make

版本控制: git

代码调试: gdb valbrind

四.源码及注释

```
#ifndef __client_H__
        #define __client_H__
        #include <pthread.h>
        #include <stdio.h>
        #include <stdlib.h>
```

```
	#include <sys/types.h>
	#include <dirent.h>

	#include "socket.h"
	#include "Queue.h"
	#include "friend.h"
	#include "listen_thread.h"
	#include "talk_thread.h"
	#include "show_thread.h"
	#include "ulib.h"
	#include "file_trans.h"


	extern socket_fd listen_socket_fd;
	extern Queue name_address;
	extern Queue connectors;
	extern Queue *file_trans_control;
	//extern pthread_t *listen_thread_id;

	int client_shutdown;//TRUE or FALSE

	void init_socket();
	int input();
	int file_mode();
	void send_file(char *friend_name, char *file_location);
	void send_message(char *friend_name, char *message);
	void send_wrap_split_data(socket_fd send_socket_fd, char *data, char tail);
	void send_split_data(socket_fd send_socket_fd, char *send_data);
	int connect_TCP_by_name(char *friend_name);
#endif /* __client_H__ */

#include "client.h"
static struct sockaddr_in listen_addr_in;

static char inputbuf[INPUT_BUFSIZE];

int main (int argc, char *argv[])
{
	client_shutdown = 0;
	init_friend_name_addr();//must be init first,load friend name reflects to address from file
	init_connector(&connectors);
	init_socket();
	init_show();
	init_file_trans_control();

	while(!client_shutdown){
		input();
	}
	shutdown(listen_socket_fd, SHUT_RDWR);
	close(listen_socket_fd);
	destroy_friend_name_addr(&name_address);
	close_all_connector(&connectors);//talk_thread can't be closed by themself because recv is
```

```
blocking
	while (connector_length(&connectors)){//wait for thread free
		usleep(50);
	}
	destroy_connector(&connectors);
	destroy_file_trans_control(file_trans_control);
	destroy_show_tty();
	sleep(1);//wait for other thread exit,not necssary but that can make it easy for valbrind check
memory leak (include still reachable)
	return 0;
}//end main-function



void init_socket()
{
	listen_socket_fd = socket(PF_INET,SOCK_STREAM,0);//PF_INET->TCP/IP Protocol
Family,SOCK_STREAM->TCP
	int optval = 1;

setsockopt(listen_socket_fd,SOL_SOCKET,SO_REUSEADDR,&optval,sizeof(SO_REUSEADDR
));//enable port multiplexing
	memset(&listen_addr_in,0,sizeof(listen_addr_in));
	listen_addr_in.sin_family = AF_INET;//AF_INET->TCP/IP Address Family
	listen_addr_in.sin_addr.s_addr = htonl(INADDR_ANY);//wildcard IPv4 address
	listen_addr_in.sin_port = htons(SERVER_PORT);
	bind(listen_socket_fd,(struct sockaddr *)&listen_addr_in,sizeof(listen_addr_in));
	listen(listen_socket_fd,LISTEN_LIST_LENGTH);//set listen list length,begin listen
	pthread_t listen_thread_id;
	pthread_create(&listen_thread_id, NULL, listen_thread, 0);//begin listen thread

}



int input(){
	printf("> ");
	memset(inputbuf, 0, INPUT_BUFSIZE);
	setbuf(stdin, NULL);
	fgets(inputbuf, INPUT_BUFSIZE, stdin);
	setbuf(stdin, NULL);
	char *friend_name = strtok(inputbuf, ":");
	char *message = strtok(NULL, "\n");
	if (friend_name != NULL && message != NULL) {
		send_message(friend_name, message);//EOF 0x04
		return TRUE;
	}

	if (!strcmp(friend_name, "exit\n")){
		client_shutdown = 1;
		return TRUE;
	}
	if (!strcmp(friend_name, "file\n")){
		file_mode();
		return TRUE;
	}

	printf("%s", "usage\t<$name><:><$message>\n");
	printf("%s", "\t<exit>\n");
	printf("%s", "\t<file>\n");

	return FALSE;
}

int file_mode(){
	printf("[file_mode]> ");
	memset(inputbuf, 0, INPUT_BUFSIZE);
	setbuf(stdin, NULL);
	fgets(inputbuf, INPUT_BUFSIZE, stdin);
	setbuf(stdin, NULL);
	char *friend_name;
	char *trans_location;
	friend_name = strtok(inputbuf, ":");
	trans_location = strtok(NULL, "\n");

	if (friend_name != NULL && trans_location != NULL) {
		send_file(friend_name, trans_location);
		return TRUE;
	}

	printf("%s", "file mode usage\t<$name><:><$location>\n");
	return FALSE;
}

void send_file(char *friend_name, char *file_location){
	//is friend exist?
	//is file exist?
	//trans file thread and connect success ;;;; must Init File Server Socket in main
	printf("[send file] %s %s\n",friend_name,file_location);



	//is dir? tar
	//else is file? get filename
	//is / in it? yes->get dir and filename
	//is / in it? no->get filename
	char *file_name;
	socket_fd friend_socket_fd;
	int file_trans_fd;
	file_name = strrchr(file_location, '/');
	if (file_name != NULL) {
		file_name++;
	}else{
		file_name = file_location;
	}
	//file_name = strtok(file_name, "\n");
	DIR *dir = opendir(file_location);
	if (dir != NULL) {
		printf("error: can't trans a directory\n");
		closedir(dir);
		return;
	}

	FILE *file = fopen(file_location, "rb");
	if (file == NULL) {
		printf("error: can't find file %s %s\n",file_location,file_name);

	}else{
		friend_socket_fd = connect_TCP_by_name(friend_name);
		if (friend_socket_fd <= 2)
			goto end;
		//register file_trans to file_trans_control
		file_trans_fd = init_file_trans(file_trans_control, TRUE, file_name, file_location, 0);


		//create talk thread file_mode
		pthread_t talk_thread_id;
		struct talk_thread_arg *tt_arg = (struct talk_thread_arg *)malloc_safe(tt_arg,
sizeof(struct talk_thread_arg));
		tt_arg->connect_socket_fd = friend_socket_fd;
		tt_arg->connect_launcher = TRUE;
		tt_arg->connect_type = FILE_CONNECT;
		tt_arg->file_trans_fd = file_trans_fd;
		pthread_create(&talk_thread_id, NULL, talk_thread, (void *)tt_arg);

		fclose(file);
	}

	end:
		return;
}

void send_message(char *friend_name, char *message){
	struct friend *this = (struct friend *)malloc_safe(this, sizeof(struct friend));

	int result = find_connector_by_name(&connectors, friend_name, this,
MESSAGE_CONNECT);//is connectted?
	if (result) {

		socket_fd friend_socket_fd;
		memset(&friend_socket_fd, 0, sizeof(socket_fd));


		//connect
		friend_socket_fd = connect_TCP_by_name(friend_name);
		if (friend_socket_fd <= 2)
}			goto end;

		//create talk thread message mode
		pthread_t talk_thread_id;
		struct talk_thread_arg *tt_arg = malloc_safe(tt_arg, sizeof(struct talk_thread_arg));
		tt_arg->connect_socket_fd = friend_socket_fd;
		tt_arg->connect_launcher = TRUE;
		tt_arg->connect_type = MESSAGE_CONNECT;
		tt_arg->file_trans_fd = -1;
		pthread_create(&talk_thread_id, NULL, talk_thread, (void *)tt_arg);

		//wait fot connect established
		while(find_connector_by_name(&connectors, friend_name,  NULL,
MESSAGE_CONNECT)){
			usleep(50);
		}
		this->friend_socket_fd = friend_socket_fd;
	}

	//send message
	send_wrap_split_data(this->friend_socket_fd, message, ETB);

	show(friend_name, message, SHOW_DIRECTION_OUT);

	end:
	free_safe(this);

}

void send_wrap_split_data(socket_fd send_socket_fd, char *data, char tail)
{
	int wrap_data_length = strlen(data) * sizeof(char) + sizeof(tail);
	char *wrap_data = (char *)malloc_string_safe(wrap_data, wrap_data_length);
	wrap(data, tail, wrap_data);
	send_split_data(send_socket_fd, wrap_data);
	free_safe(wrap_data);

}

void send_split_data(socket_fd send_socket_fd, char *send_data)
{
	int send_data_length = strlen(send_data) + 2;
	for (int i = 0; i <= send_data_length / SEND_BUFSIZE; i += 1) {
		char *sendbuf = (char *)malloc_string_safe(sendbuf, SEND_BUFSIZE *
sizeof(char));
		strncpy(sendbuf, send_data + i * SEND_BUFSIZE, SEND_BUFSIZE);
		send(send_socket_fd, sendbuf, strlen(sendbuf), 0);
		free_safe(sendbuf);
	}
}
```

```
int connect_TCP_by_name(char *friend_name)
{
	int connect_socket_fd = socket(PF_INET, SOCK_STREAM, 0);//PF_INET->TCP/IP
Protocol Family,SOCK_STREAM->TCP
	//init sock addr
	struct sockaddr_in dest_addr;
	char friend_ip[16] = {0};
	int gfa_result = get_friend_address(&name_address , friend_name, (char *)&friend_ip);
	if (gfa_result)
		return ERROR;
	dest_addr.sin_family = AF_INET;//AF_INET->TCP/IP Address Family
	dest_addr.sin_port = htons(SERVER_PORT);
	dest_addr.sin_addr.s_addr = inet_addr((char *)&friend_ip);
	memset(&(dest_addr.sin_zero), 0, sizeof(dest_addr.sin_zero));

	int result = connect(connect_socket_fd, (struct sockaddr *)&dest_addr, sizeof(struct
sockaddr));
	if (result == -1) //connect failed
		return ERROR;

	return connect_socket_fd;
}
#include "file_trans.h"
static pthread_rwlock_t file_trans_control_rwlock;

Queue *init_file_trans_control()
{
	pthread_rwlock_init(&file_trans_control_rwlock, NULL);
	file_trans_control = (Queue *)malloc_safe(file_trans_control, sizeof(Queue));
	InitQueue(file_trans_control, sizeof(struct file_trans));
	return file_trans_control;
}

int file_trans_control_length(Queue *file_trans_control)
{
	pthread_rwlock_rdlock(&file_trans_control_rwlock);
	int length = QueueLength(file_trans_control);
	pthread_rwlock_unlock(&file_trans_control_rwlock);
	return length;
}

void destroy_file_trans_control(Queue *file_trans_control)
{
	pthread_rwlock_rdlock(&file_trans_control_rwlock);
	QNode *p = file_trans_control->front;
	while((p = p->next)){
		struct file_trans *task = (struct file_trans *)p->pointer;

		destroy_file_trans(file_trans_control, task->file_trans_fd);
	}
	pthread_rwlock_unlock(&file_trans_control_rwlock);
	DestroyQueue(file_trans_control);
	free_safe(file_trans_control);
}



/*int init_file_trans(Queue *file_trans_control, int connect_launcher, char *file_name, char
*file_location, char *md5)*/
int init_file_trans(Queue *file_trans_control, int connect_launcher, char *file_name, char
*file_location, long total_size)
{
	struct file_trans *task = (struct file_trans *)malloc_safe(task, sizeof(struct file_trans));

	pthread_rwlock_rdlock(&file_trans_control_rwlock);
	//init file_trans_control
	if (file_trans_control->front->next != NULL)
		task->file_trans_fd = ((struct file_trans *)file_trans_control->front->next->pointer)-
>file_trans_fd + 1;
	else
		task->file_trans_fd = 0;
	pthread_rwlock_unlock(&file_trans_control_rwlock);
	//init connect_launcher --launcher(TRUE) -- trans direction out; recivier(FALSE) -- trans
direction in
	task->connect_launcher = connect_launcher;

	//init file_name
	task->file_name = (char *)malloc_string_safe(task->file_name, strlen(file_name) *
sizeof(char));
	strncpy(task->file_name, file_name, strlen(file_name));

	//init file_location md5 file_ptr total_size
	if (connect_launcher == TRUE) {//launcher
		task->file_location = (char *)malloc_string_safe(task->file_location,
strlen(file_location) * sizeof(char));
		strncpy(task->file_location, file_location, strlen(file_location));
/*		task->md5 = init_md5(file_location);*/
/*		printf("[md5]%s\n",task->md5);*/
		task->file_ptr = fopen(file_location, "rb");
		fseek(task->file_ptr, 0, SEEK_END);
		task->total_size = ftell(task->file_ptr);
		fseek(task->file_ptr, 0, SEEK_SET);
	}else{//receiver
		task->file_location = (char *)malloc_string_safe(task->file_location,
(strlen(DOWNLOAD_PATH) + strlen(file_name)) * sizeof(char));

		strncpy(task->file_location, DOWNLOAD_PATH, strlen(DOWNLOAD_PATH));
		strncpy(task->file_location + strlen(DOWNLOAD_PATH), file_name,
strlen(file_name));
		task->file_ptr = fopen(task->file_location, "wb");
		task->total_size = total_size;
	}

	//init total_size
	task->fin_size = 0;
	pthread_rwlock_wrlock(&file_trans_control_rwlock);
	EnQueue(file_trans_control, task);
	pthread_rwlock_unlock(&file_trans_control_rwlock);
	return task->file_trans_fd;
}

struct file_trans *find_file_trans_task(Queue *file_trans_control, int file_trans_fd)
{
	struct file_trans *task = NULL;
	pthread_rwlock_wrlock(&file_trans_control_rwlock);
	QNode *p = file_trans_control->front;
	while((p = p->next)){
		task = (struct file_trans *)p->pointer;
		if (file_trans_fd == task->file_trans_fd) {
			pthread_rwlock_unlock(&file_trans_control_rwlock);
			return task;
		}
	}
	pthread_rwlock_unlock(&file_trans_control_rwlock);
	return task;
}



void destroy_file_trans(Queue *file_trans_control, int file_trans_fd)
{
	pthread_rwlock_wrlock(&file_trans_control_rwlock);
	QNode *p = file_trans_control->front;
	QNode *before = p;
	while((p = p->next)){
		struct file_trans *task = (struct file_trans *)p->pointer;
		if (file_trans_fd == task->file_trans_fd) {
			before->next = p->next;
			if (file_trans_control->rear == p)
				file_trans_control->rear = file_trans_control->front;

			free_safe(task->file_name);
			free_safe(task->file_location);
			fclose(task->file_ptr);
			free_safe(p->pointer);
			free_safe(p);
			pthread_rwlock_unlock(&file_trans_control_rwlock);
			return;
		}
		before = p;
	}
	pthread_rwlock_unlock(&file_trans_control_rwlock);
	return;
}

#ifndef __file_trans_H__
```

```
	#define __file_trans_H__
	#include <stdio.h>
	#include <pthread.h>
	#include <errno.h>
	#include "socket.h"
	#include "client.h"
	#include "Queue.h"

	#define DOWNLOAD_PATH "./DP2PCSDownload/"
	#define SIZE_INFO_HEAD "size is "
	#define SIZE_INFO_TAIL " bytes"

	struct file_trans{
		int file_trans_fd;//set by file trans control
		int connect_launcher;//set when init ---- TRUE is launcher;FALSE is the accpeter
		char *file_name;//set when init
		char *file_location;//Launcher -- set when init; Recevier -- set by itself
		FILE *file_ptr;//get by itself# Recevier write binary; Launcher read binary
		long total_size;//Recevier --set when init; Launcher -- get by itself
		long fin_size;//set when trans happen
	};

	extern int client_shutdown;

	Queue *file_trans_control;

	Queue *init_file_trans_control();
	int file_trans_control_length(Queue *file_trans_control);
	void destroy_file_trans_control(Queue *file_trans_control);


	int init_file_trans(Queue *file_trans_control, int connect_launcher, char *file_name, char
*file_location, long total_size);
	struct file_trans *find_file_trans_task(Queue *file_trans_control, int file_trans_fd);
	void destroy_file_trans(Queue *file_trans_control, int file_trans_fd);

	int read_file_trans_block(Queue *file_trans_control, int file_trans_fd, unsigned char
*file_block);
	void append_file_trans_block(Queue *file_trans_control, int file_trans_fd, unsigned char
*block, int write_size);
	int send_file_trans_block(Queue *file_trans_control, int file_trans_fd, socket_fd
connect_socket_fd, unsigned char *file_block);
	int recv_file_trans_block(Queue *file_trans_control, int file_trans_fd, socket_fd
connect_socket_fd, unsigned char *file_block);

#endif /* __file_trans_H__ */
#include "friend.h"
static pthread_rwlock_t name_addr_rwlock;
static pthread_rwlock_t connector_rwlock;
void init_friend_name_addr()
{
	pthread_rwlock_init(&name_addr_rwlock, NULL);
	FILE *friend_address_file;
	friend_address_file = fopen(FRIEND_ADDRESS_FILE,"r");
	InitQueue(&name_address, sizeof(struct friend_name_addr));
	char line_data[LINE_LENGTH] = {0};
	while(fgets(line_data,LINE_LENGTH,friend_address_file) != NULL){
		char *friend_name;
		char *friend_address;
		friend_name = strtok(line_data, "@");
		friend_address = strtok(NULL, "\n");
		if(friend_name == NULL || friend_address == NULL)
			continue;
		enqueue_friend_name_addr(&name_address, friend_name, friend_address);
		memset(line_data, 0, sizeof(line_data));
	}
	fclose(friend_address_file);
}

void enqueue_friend_name_addr(LinkQueue *queue, char *friend_name,char *friend_address)
{
	struct friend_name_addr *fna;
	fna = (struct friend_name_addr *)malloc_safe(fna, sizeof(struct friend_name_addr));
	fna->friend_name = (char *)malloc_string_safe(fna->friend_name, strlen(friend_name) *
sizeof(char));
	fna->friend_address = (char *)malloc_string_safe(fna->friend_address,
strlen(friend_address) * sizeof(char));
	strcpy(fna->friend_name, friend_name);
	strcpy(fna->friend_address, friend_address);
	pthread_rwlock_wrlock(&name_addr_rwlock);
	EnQueue(queue, (void *)fna);
	pthread_rwlock_unlock(&name_addr_rwlock);
	free_safe(fna);
}

void dequeue_friend_name_addr(LinkQueue *queue)
{
	struct friend_name_addr *fna;
	fna = (struct friend_name_addr *)malloc_safe(fna, sizeof(struct friend_name_addr));
	pthread_rwlock_wrlock(&name_addr_rwlock);
	DeQueue(queue, (void *)fna);
	pthread_rwlock_unlock(&name_addr_rwlock);
	free_safe(fna->friend_name);
	free_safe(fna->friend_address);
	free_safe(fna);
}

void destroy_friend_name_addr(LinkQueue *queue)
{
	while(QueueLength(queue) != 0){
		dequeue_friend_name_addr(queue);
	}
	DestroyQueue(queue);
	pthread_rwlock_destroy(&name_addr_rwlock);
```

```
}

int get_friend_address(LinkQueue *name_address_queue, char *friend_name, char *friend_ip)
{
	pthread_rwlock_rdlock(&name_addr_rwlock);
	QNode *p = name_address_queue->front;
	while((p = p->next)){
		if (!strcmp(friend_name, ((struct friend_name_addr *)p->pointer)->friend_name)) {
			strcpy(friend_ip, ((struct friend_name_addr *)p->pointer)->friend_address);
			pthread_rwlock_unlock(&name_addr_rwlock);
			return OK;
		}
	}
	pthread_rwlock_unlock(&name_addr_rwlock);
	return ERROR;
}

int get_friend_name(LinkQueue *name_address_queue, char *friend_ip, char *friend_name)
{
	pthread_rwlock_rdlock(&name_addr_rwlock);
	QNode *p = name_address_queue->front;
	while((p = p->next)){
		if (!strcmp(friend_ip, ((struct friend_name_addr *)p->pointer)->friend_address)) {
			strcpy(friend_name, ((struct friend_name_addr *)p->pointer)->friend_name);
			pthread_rwlock_unlock(&name_addr_rwlock);
			return OK;
		}
	}
	pthread_rwlock_unlock(&name_addr_rwlock);
	return ERROR;
}

int get_friend_name_length(LinkQueue *name_address_queue, char *friend_ip)
{
	pthread_rwlock_rdlock(&name_addr_rwlock);
	QNode *p = name_address_queue->front;
	while((p = p->next)){
		if (!strcmp(friend_ip, ((struct friend_name_addr *)p->pointer)->friend_address)) {
			pthread_rwlock_unlock(&name_addr_rwlock);
			return strlen(((struct friend_name_addr *)p->pointer)->friend_name);
		}
	}
	pthread_rwlock_unlock(&name_addr_rwlock);
	return 0;
}

int init_connector(LinkQueue *friend_queue)
{
	pthread_rwlock_init(&connector_rwlock, NULL);
	int result = InitQueue(friend_queue, sizeof(struct friend));
	return result;
}
```

```
int enqueue_connector(LinkQueue *friend_queue, char *friend_name, pthread_t friend_thread_id,
socket_fd friend_socket_fd, int connect_type)
{
	struct friend *connector = (struct friend *)malloc_safe(connector, sizeof(struct friend));
	connector->friend_name = (char *)malloc_string_safe(connector->friend_name,
strlen(friend_name) * sizeof(char));
	strcpy(connector->friend_name, friend_name);
	connector->friend_thread_id = friend_thread_id;
	connector->friend_socket_fd = friend_socket_fd;
	connector->connect_type = connect_type;
	pthread_rwlock_wrlock(&connector_rwlock);
	int result = EnQueue(friend_queue, (void *)connector);
	pthread_rwlock_unlock(&connector_rwlock);
	free_safe(connector);
	return result;
}

int dequeue_connector_length(LinkQueue *friend_queue)
{
	pthread_rwlock_rdlock(&connector_rwlock);
	int result = strlen(((struct friend *)friend_queue->front->next->pointer)->friend_name);
	pthread_rwlock_unlock(&connector_rwlock);
	return result;
}


int dequeue_connector(LinkQueue *friend_queue, struct friend *friend_val)
{
	struct friend *connector = (struct friend *)malloc_safe(connector, sizeof(struct friend));

	pthread_rwlock_wrlock(&connector_rwlock);
	int result = DeQueue(friend_queue, (void *)connector);
	pthread_rwlock_unlock(&connector_rwlock);
	if(friend_val != NULL){
		friend_val->friend_socket_fd = connector->friend_socket_fd;
		friend_val->friend_thread_id = connector->friend_thread_id;
		strcpy(friend_val->friend_name, connector->friend_name);
	}
	free_safe(connector->friend_name);
	free_safe(connector);
	return result;
}

int find_connector_by_name(LinkQueue *friend_queue, char *friend_name, struct friend
*friend_val, int connect_type)
{
	pthread_rwlock_rdlock(&connector_rwlock);

	QNode *p = friend_queue->front;
	while((p = p->next)){
		if (!strcmp(friend_name, ((struct friend *)p->pointer)->friend_name) && (((struct
friend *)p->pointer)->connect_type == connect_type)) {
			if (friend_val != NULL)
				memcpy(friend_val, p->pointer, sizeof(struct friend));
			pthread_rwlock_unlock(&connector_rwlock);
			return OK;
		}
	}
	pthread_rwlock_unlock(&connector_rwlock);
	return ERROR;
}

int find_connector_by_threadid(LinkQueue *friend_queue, pthread_t friend_thread_id, struct friend
*friend_val)
{
	pthread_rwlock_rdlock(&connector_rwlock);
	QNode *p = friend_queue->front;
	while((p = p->next)){
		if (!memcmp(&friend_thread_id, &((struct friend *)p->pointer)->friend_thread_id,
sizeof(pthread_t))) {
			if (friend_val != NULL)
				memcpy(friend_val, p->pointer, sizeof(struct friend));
			pthread_rwlock_unlock(&connector_rwlock);
			return OK;
		}
	}
	pthread_rwlock_unlock(&connector_rwlock);
	return ERROR;
}

int connector_length(LinkQueue *friend_queue)
{
	pthread_rwlock_rdlock(&connector_rwlock);
	int length = QueueLength(friend_queue);
	pthread_rwlock_unlock(&connector_rwlock);
	return length;
}

int remove_connector(LinkQueue *friend_queue, socket_fd talk_socket_fd)
{
	pthread_rwlock_wrlock(&connector_rwlock);
	QNode *p = friend_queue->front;
	QNode *before = p;
	while((p = p->next)){
		if (talk_socket_fd == ((struct friend *)p->pointer)->friend_socket_fd) {
			before->next = p->next;
			if (friend_queue->rear == p)
				friend_queue->rear = friend_queue->front;
			free_safe(((struct friend *)p->pointer)->friend_name);
			free_safe(p->pointer);
			free_safe(p);
			pthread_rwlock_unlock(&connector_rwlock);
			return OK;
			}
			before = p;
		}
		pthread_rwlock_unlock(&connector_rwlock);
		return ERROR;
}

void close_connector(socket_fd talk_socket_fd)
{
		shutdown(talk_socket_fd, SHUT_RDWR);
		close(talk_socket_fd);
}

void close_all_connector(LinkQueue *friend_queue)
{
		pthread_rwlock_rdlock(&connector_rwlock);
		QNode *p = friend_queue->front;
		while((p = p->next)){
			close_connector(((struct friend *)p->pointer)->friend_socket_fd);
		}
		pthread_rwlock_unlock(&connector_rwlock);
}

void destroy_connector(LinkQueue *friend_queue)
{
		pthread_rwlock_wrlock(&connector_rwlock);
		while(QueueLength(friend_queue) != 0){
			dequeue_connector(friend_queue, NULL);
		}
		DestroyQueue(friend_queue);
		pthread_rwlock_unlock(&connector_rwlock);
		pthread_rwlock_destroy(&connector_rwlock);
}

#ifndef __friend_H__
		#define __friend_H__

		#include "Queue.h"
		#include "socket.h"
		#include <pthread.h>
		#include <stdio.h>
		#include <unistd.h>


		#define FRIEND_ADDRESS_FILE "friend_address"
		#define LINE_LENGTH 100



		struct friend_name_addr{
			char *friend_name;
			char *friend_address;};
		Queue name_address;
```

```
	struct friend{
		char *friend_name;
		pthread_t friend_thread_id;
		socket_fd friend_socket_fd;
		int connect_type;};//MESSAGE_CONNECT FILE_CONNECT
	Queue connectors;



	void init_friend_name_addr();
	void enqueue_friend_name_addr(LinkQueue *queue, char *friend_name,char
*friend_address);
	void dequeue_friend_name_addr(LinkQueue *queue);
	void destroy_friend_name_addr(LinkQueue *queue);
	int get_friend_address(LinkQueue *name_address_queue, char *friend_name, char
*friend_ip);
	int get_friend_name(LinkQueue *name_address_queue, char *friend_ip, char
*friend_name);
	int get_friend_name_length(LinkQueue *name_address_queue, char *friend_ip);

	int init_connector();
	int enqueue_connector(LinkQueue *friend_queue, char *friend_name, pthread_t
friend_thread_id, socket_fd friend_socket_fd, int connect_type);
	int dequeue_connector_length(LinkQueue *friend_queue);
	int dequeue_connector(LinkQueue *friend_queue, struct friend *friend_val);
	int find_connector_by_name(LinkQueue *friend_queue, char *friend_name, struct friend
*friend_val, int connect_type);
	int find_connector_by_threadid(LinkQueue *friend_queue, pthread_t friend_thread_id,
struct friend *friend_val);
	int connector_length(LinkQueue *friend_queue);
	int remove_connector(LinkQueue *friend_queue, socket_fd talk_socket_fd);
	void close_connector(socket_fd talk_socket_fd);
	void close_all_connector(LinkQueue *friend_queue);
	void destroy_connector(LinkQueue *friend_queue);
#endif /* __friend_H__ */


#include "listen_thread.h"

void *listen_thread(void *arg)
{
	pthread_detach(pthread_self());
	while (!client_shutdown) {
		socket_fd talk_socket_fd;
		struct sockaddr client_addr_in;
		socklen_t client_addr_in_len = sizeof(struct sockaddr);
		talk_socket_fd = accept(listen_socket_fd,
						(struct sockaddr *)&client_addr_in,
						&client_addr_in_len);
		if (talk_socket_fd > 2) {//0 stdin; 1 stdout; 2 stderr
```

```
				pthread_t talk_thread_id;
				struct talk_thread_arg *tt_arg = malloc_safe(tt_arg, sizeof(struct
talk_thread_arg));
				tt_arg->connect_socket_fd = talk_socket_fd;
				tt_arg->connect_launcher = FALSE;
				tt_arg->file_trans_fd = ERROR;
				pthread_create(&talk_thread_id, NULL, talk_thread, (void *)tt_arg);
			}else{
				usleep(500);
			}
		}
		pthread_exit((void *)NULL);
}



#ifndef __listen_thread_H__
	#define __listen_thread_H__

	#include <pthread.h>
	#include <unistd.h>
	#include "socket.h"
	#include "talk_thread.h"
	#include "client.h"

	extern socket_fd listen_socket_fd;
	extern int client_shutdown;
	void *listen_thread(void *arg);

#endif /* __listen_thread_H__ */
#include "Queue.h"

int InitQueue(LinkQueue *Q, size_t value_size){
	Q->front = Q->rear = (QueuePtr)malloc_safe(Q->front, sizeof(QNode));
	Q->front->pointer = (void*)malloc_safe(Q->front->pointer, value_size);

	if(!Q->front)return ERROR;
	Q->value_size=value_size;
	Q->front->next=NULL;
	return OK;
}

int EnQueue(LinkQueue *Q, void *pointer){
	QNode *p = (QueuePtr)malloc_safe(p, sizeof(QNode));//malloc 由编译器分配空间的间隔会
自动大于等于 20!!
	if(!p)return ERROR;
	p->pointer=(void*)malloc_safe(p->pointer, Q->value_size);
	if(!p->pointer)return ERROR;
	memcpy(p->pointer,pointer,Q->value_size);//p->pointer=pointer;
	p->next=NULL;
	Q->rear->next=p;
	Q->rear=p;
	return OK;
}

int DeQueue(LinkQueue *Q, void *pointer){
	QNode *p;
	if(Q->front==Q->rear){return ERROR;}
	p=Q->front->next;
	memcpy(pointer,p->pointer,Q->value_size);//pointer=p->pointer;
	Q->front->next=p->next;
	if(Q->rear==p)
		Q->rear=Q->front;
	free_safe(p->pointer);
	free_safe(p);
	return OK;
}

int DestroyQueue(LinkQueue *Q){
	while (Q->front){
		Q->rear = Q->front->next;
		free_safe(Q->front->pointer);
		free_safe(Q->front);
		Q->front=Q->rear;
	}//end while
	return OK;
}

int QueueLength(LinkQueue *Q){
	QNode *p=Q->front;
	int length=0;
	while((p = p->next)){
		length++;
	}
	return length;
}
#ifndef __Queue_H__
	#define __Queue_H__
	#include <stdio.h>
	#include <malloc.h>
	#include <memory.h>

	#include "ulib.h"

	#ifndef ERROR
	#define ERROR -1
	#endif

	#ifndef OK
	#define OK 0
	#endif

	typedef struct QNode{  /*队列的链节点*/
		struct QNode *next;
```

```
                void* pointer;//指向
        }QNode,*QueuePtr;

        typedef struct{
                QueuePtr front;//队头指针
                QueuePtr rear;//队尾指针
                //size_t ptr_size;//队列节点指向的数据块的指针大小
                size_t value_size;//队列节点指向的数据块的大小
        }LinkQueue,Queue;

        int InitQueue(LinkQueue *Q, size_t value_size);
        int EnQueue(LinkQueue *Q, void* pointer);
        int DeQueue(LinkQueue *Q, void* pointer);
        int DestroyQueue(LinkQueue *Q);
        int QueueLength(LinkQueue *Q);
#endif /* __Queue_H__ */
#include "show_thread.h"

void show(char *friend_name, char *message, int direction)
{

        time_t time_now;
        time(&time_now);
        struct tm tmn;
        localtime_r(&time_now, &tmn);
        int show_string_len = sizeof(struct tm) + sizeof(LIGHT_RED) + strlen(friend_name) *
sizeof(char) + sizeof(COLOR_NONE) + strlen(message) * sizeof(char) + 13 * sizeof(char);//here
13 is the length of ':' '@' ' ' \n and \t counts sum
        char *show_string = (char *)malloc_string_safe(show_string, show_string_len);
     if (direction == SHOW_DIRECTION_IN) {
        sprintf(show_string, "%d-%d-%d %d:%d:%d %s%s%s:\n\t%s\n", (&tmn)->tm_year+1900,
(&tmn)->tm_mon+1, (&tmn)->tm_mday, (&tmn)->tm_hour, (&tmn)->tm_min, (&tmn)->tm_sec,
LIGHT_RED, friend_name,COLOR_NONE, message);
     }else if (direction == SHOW_DIRECTION_OUT) {
                sprintf(show_string, "%d-%d-%d %d:%d:%d @%s%s%s:\n\t%s\n", (&tmn)-
>tm_year+1900, (&tmn)->tm_mon+1, (&tmn)->tm_mday, (&tmn)->tm_hour, (&tmn)->tm_min,
(&tmn)->tm_sec, LIGHT_BLUE, friend_name,COLOR_NONE, message);
     }else if (direction == SHOW_DIRECTION_SYSTEM_INFO) {
        sprintf(show_string, "[info] %s %s\n", friend_name, message);
     }


     int command_length = show_string_len + (strlen("echo \" \">>") + strlen(show_tty_running-
>show_tty_name)) * sizeof(char);
     char *command = (char *)malloc_string_safe(command, command_length);
     sprintf(command, "%s%s%s%s", "echo \"", show_string, "\">>", show_tty_running-
>show_tty_name);
        system(command);
        free_safe(show_string);
        free_safe(command);
}
```

```
void init_show()
{
	show_tty_running = (struct show_tty*)malloc_safe(show_tty_running, sizeof(struct
show_tty));
	show_tty_running->show_tty_name = (char *)malloc_string_safe(show_tty_running-
>show_tty_name, SHOW_TTY_NAME_BUFSIZE);
	system("bash show_tty_daemon.sh show");
	refresh_show_tty();
	pthread_t show_thread_id;
	pthread_create(&show_thread_id, NULL, show_thread, 0);
}

void refresh_show_tty()
{
	FILE *file = fopen(SHOW_TTY_FILE,"r");
	if (file == NULL) {
		return;
	}

	int ttyname_length;
	fseek(file, 0L, SEEK_END);
	ttyname_length = ftell(file);
	char *show_tty_name = (char *)malloc_string_safe(show_tty_name, ttyname_length *
sizeof(char));
	char *show_tty_pidbuf = (char *)malloc_string_safe(show_tty_pidbuf, ttyname_length *
sizeof(char));

	fseek(file, 0L, SEEK_SET);
	fscanf(file, "%s %s", show_tty_pidbuf, show_tty_name);


	show_tty_running->show_tty_pid = atoi(show_tty_pidbuf);
	memset(show_tty_running->show_tty_name, 0, SHOW_TTY_NAME_BUFSIZE);
	memcpy(show_tty_running->show_tty_name, show_tty_name,strlen(show_tty_name) *
sizeof(char));
	free_safe(show_tty_pidbuf);
	free_safe(show_tty_name);
	fclose(file);
}

void *show_thread(void *arg)
{
	pthread_detach(pthread_self());
	sleep(3);//wait for ensure no conflict with create tty in init method
	while(!client_shutdown){
		int is_tty_reboot = 0;
		is_tty_reboot = system("bash show_tty_daemon.sh isalive");
		if (is_tty_reboot) {//if a new tty created and olds are killed
			refresh_show_tty();
		}
		sleep(1);
	}


/*	pthread_mutex_destroy(&lock);*/

	pthread_exit((void *)NULL);
}

void destroy_show_tty()
{
	free_safe(show_tty_running->show_tty_name);
	free_safe(show_tty_running);
	system("bash show_tty_daemon.sh killsame");
}
#ifndef __show_thread_H__
	#define __show_thread_H__

	#include <stdio.h>
	#include <stdlib.h>
	#include <unistd.h>
	#include <time.h>
	//#include <pthread.h>
	#include <string.h>

	#include "client.h"

	#define COLOR_NONE "\033[m"
	#define LIGHT_RED "\033[1;31m"
	#define LIGHT_BLUE "\033[1;34m"

	#define SHOW_TTY_FILE "show_tty_name.txt"
	#define SHOW_TTY_NAME_BUFSIZE 50
	#define SHOW_DIRECTION_IN 0
	#define SHOW_DIRECTION_OUT 1
	#define SHOW_DIRECTION_SYSTEM_INFO 3
	extern int client_shutdown;

	struct show_tty{
		int show_tty_pid;
		char *show_tty_name;
	};
	struct show_tty *show_tty_running;

	void show(char *friend_name, char *message, int dirction);
	void init_show();
	void *show_thread(void *arg);
	void refresh_show_tty();
	void destroy_show_tty();
#endif /* __show_thread_H__ */
#!/bin/bash
currentpath=`dirname $0`
search='show_tty_name.txt;tty>>show_tty_name.txt;while true;do read -s;done'
```

```
function isalive(){
	local pid=`ps -ef|grep "${search}"|grep -v "grep"|awk '{print $2}'`
	local pidArray=($pid)
	local pidNum=`ps -ef|grep "${search}"|grep -v "grep"|awk '{print $2}'|wc -l`
	#echo "pid="$pid
	if [ ${pidNum} -gt 1 ] ; then
		killsame
		#gnome-terminal -t "Chatting" -x bash -c "bash ${currentpath}/show_tty_daemon.sh 
show"
		bash ${currentpath}/show_tty_daemon.sh show
		return 1
	elif [ ${pidNum} -eq 0 ]; then
		#gnome-terminal -t "Chatting" -x bash -c "bash ${currentpath}/show_tty_daemon.sh 
show"
		bash ${currentpath}/show_tty_daemon.sh show
		return 1
	elif [ ${pidNum} -eq 1 ];then
		return 0
	fi
}

function killsame(){
	local pid=`ps -ef|grep "${search}"|grep -v "grep"|awk '{print $2}'`
	local pidArray=($pid)
	local pidNum=`ps -ef|grep "${search}"|grep -v "grep"|awk '{print $2}'|wc -l`
	#echo "pid="$pid
	if [ ${#pid} -ne 0 ] ; then
		for (( i = 0; i < pidNum; i++)); do
			exec kill -9 ${pidArray[i]} &
			#echo ${pidArray[i]}
		done;
	fi
}

function show(){
	gnome-terminal -t "Chatting" -x bash -c "echo -ne '\033[?25l';echo \"$
$\">show_tty_name.txt;tty>>show_tty_name.txt;while true;do read -s;done"
}



if [ $# -eq 1 ]; then
	if [ "$1" == "killsame" ]; then
		killsame
	elif [ "$1" == "isalive" ]; then
		isalive
	elif [ "$1" == "show" ]; then
		show
	fi
fi
```

```
21093
/dev/pts/26
#ifndef __socket_H__
	#define __socket_H__

	#include <netinet/in.h>
	#include <sys/types.h>
	#include <sys/socket.h>
	#include <arpa/inet.h>

	typedef int socket_fd;
	socket_fd listen_socket_fd;

	#define SERVER_PORT 12345
	#define RECV_BUFSIZE 10
	#define SEND_BUFSIZE RECV_BUFSIZE
	#define INPUT_BUFSIZE 1024*4
	#define LISTEN_LIST_LENGTH 20

	#define SOH '\x1'	//start of headline
	#define STX '\x02'	//start of text
	#define EOT '\x04'	//end of transmission
	#define ENQ '\x05'	//enquiry
	#define ACK '\x06'	//acknowledge
	#define NAK '\x15'	//negative acknowledge
	#define ETB '\x17'	//end of trans. block
	#define CAN '\x18'	//cancel

	#define ACK_STR "\x06\x17"
	#define EOT_STR "\x04\x17"
	#define CAN_STR "\x18\x17"

		//connect type
	#define MESSAGE_CONNECT 0
	#define MESSAGE_CONNECT_STR "0"
	#define FILE_CONNECT 1
	#define FILE_CONNECT_STR "1"


#endif /* __socket_H__ */
#include "talk_thread.h"

void *talk_thread(void *arg)
{

	pthread_detach(pthread_self());
	pthread_t friend_thread_id = pthread_self();
	//socket_fd ==> address ==> friend_name enqueue_connector
	//get socket_fd
	socket_fd talk_socket_fd = ((struct talk_thread_arg *)arg)->connect_socket_fd;
	//int state = TALK_RUNNING;
	int connect_launcher = ((struct talk_thread_arg *)arg)->connect_launcher;
	int connect_type;
	int file_trans_fd = (int)((struct talk_thread_arg *)arg)->file_trans_fd;


	//can create a new function get_friend_address(socket_fd talk_socket_fd, char *ip)
	struct sockaddr addr;
	socklen_t addr_len = sizeof(struct sockaddr);
	getpeername(talk_socket_fd, (struct sockaddr *)&addr, &addr_len);

	//get address
	char ip[16] = {0};
	strcpy(ip, inet_ntoa(((struct sockaddr_in *)&addr)->sin_addr));

	char *friend_name = (char *)malloc_string_safe(friend_name,
get_friend_name_length(&name_address, ip) * sizeof(char));

	get_friend_name(&name_address, ip, friend_name);
	//get friend_name

	//init data recv
	Queue *data_recv = init_split_data_recv();
	if (talk_socket_fd == 0)
		goto end;

	//set connect type
	if (connect_launcher == FALSE) {//for receiver	recv connect type, set type, send ACK
		//recv head control data_length
		if (recv_unwrap_split_data(talk_socket_fd, data_recv, NULL) == FALSE) goto end;
		char *head_control_data = init_data_recombine(data_recv);
		recombine_data(data_recv, head_control_data);
		connect_type = atoi(head_control_data);
		destroy_data_recombine(head_control_data);
		send(talk_socket_fd, ACK_STR, strlen(ACK_STR), 0);
	}else{//for launcher	get and set connect type, send type, recv ACK(no need to check is
data equal ACK)
		connect_type = ((struct talk_thread_arg *)arg)->connect_type;
		char *send_control_str;
		if (connect_type == MESSAGE_CONNECT)
			send_control_str = MESSAGE_CONNECT_STR;
		if (connect_type == FILE_CONNECT)
			send_control_str = FILE_CONNECT_STR;
		//send head control data
		send_wrap_split_data(talk_socket_fd, send_control_str, ETB);
		//recv head control ACK
		if (recv_equal_char(talk_socket_fd, ACK) == FALSE) goto end;

	}

	//enqueue_connector
	enqueue_connector(&connectors, friend_name, friend_thread_id, talk_socket_fd,
connect_type);
```

```
	//init and set arg by type
	//runs service
	//destroy
	struct connect_info *cinfo;
	if (connect_type == MESSAGE_CONNECT) {
		cinfo = init_message(talk_socket_fd, friend_name, data_recv);
		show_message(cinfo);
		destroy_message(cinfo);
	}else if (connect_type == FILE_CONNECT) {
		cinfo = init_download(talk_socket_fd, friend_name, data_recv, file_trans_fd);
		download_file(cinfo);
		destroy_download(cinfo);
	}else{
		goto end;
	}


	end:
	free_safe(arg);
	destroy_split_data_recv(data_recv);
	if (!find_connector_by_threadid(&connectors, friend_thread_id, NULL)) {
		remove_connector(&connectors, talk_socket_fd);
	}
	close_connector(talk_socket_fd);
	free_safe(friend_name);
	pthread_exit((void *)NULL);
}


struct connect_info *init_message(socket_fd talk_socket_fd, char *friend_name, Queue *data_recv)
{
	struct connect_info *cinfo = (struct connect_info*)malloc_safe(cinfo, sizeof(struct
connect_info));
	cinfo->data_recv = data_recv;
	cinfo->connect_socket_fd = talk_socket_fd;
	cinfo->friend_name = friend_name;
	return cinfo;
}


void show_message(struct connect_info *cinfo)
{
	while (!client_shutdown) {
		if (recv_unwrap_split_data(cinfo->connect_socket_fd, cinfo->data_recv, NULL) ==
FALSE) break;
		char *message = init_data_recombine(cinfo->data_recv);
		recombine_data(cinfo->data_recv, message);
		show(cinfo->friend_name, message, SHOW_DIRECTION_IN);
		destroy_data_recombine(message);
		usleep(500);
	}
```

```
}
void destroy_message(struct connect_info *cinfo)
{
	free_safe(cinfo);
}


struct connect_info *init_download(socket_fd talk_socket_fd, char *friend_name, Queue
*data_recv, int file_trans_fd)
{
	struct connect_info *cinfo = (struct connect_info*)malloc_safe(cinfo, sizeof(struct
connect_info));
	struct file_trans *task;

	if (file_trans_fd == ERROR) {//Accepter recv file_name init
		if (recv_unwrap_split_data(talk_socket_fd, data_recv, NULL) == FALSE) return
NULL;
		char *file_name = init_data_recombine(data_recv);
		recombine_data(data_recv, file_name);
		send(talk_socket_fd, ACK_STR, strlen(ACK_STR), 0);

		if (recv_unwrap_split_data(talk_socket_fd, data_recv, NULL) == FALSE) return
NULL;
		char *total_size_str = init_data_recombine(data_recv);
		recombine_data(data_recv, total_size_str);

		send(talk_socket_fd, ACK_STR, strlen(ACK_STR), 0);
		long total_size = atol(total_size_str);
		file_trans_fd = init_file_trans(file_trans_control, FALSE, file_name, NULL,
total_size);
		task = find_file_trans_task(file_trans_control, file_trans_fd);
		char *size_info = (char *)malloc_string_safe(size_info, strlen(SIZE_INFO_HEAD)
+ strlen(total_size_str) + strlen(SIZE_INFO_TAIL));
		sprintf(size_info, "%s%s%s", SIZE_INFO_HEAD, total_size_str,
SIZE_INFO_TAIL);
		show(task->file_name, size_info, SHOW_DIRECTION_SYSTEM_INFO);
		destroy_data_recombine(file_name);
		destroy_data_recombine(total_size_str);
		free(size_info);
/*		destroy_data_recombine(md5);*/

	}else{//Launcher send file_name
		task = find_file_trans_task(file_trans_control, file_trans_fd);

		send_wrap_split_data(talk_socket_fd, task->file_name, ETB);
		//recv head control ACK
		if (recv_equal_char(talk_socket_fd, ACK) == FALSE) return NULL;

		//send total_size_str
		char *send_total_size_str = long_to_string(task->total_size);

		char *size_info = (char *)malloc_string_safe(size_info, strlen(SIZE_INFO_HEAD)
+ strlen(send_total_size_str) + strlen(SIZE_INFO_TAIL));
                sprintf(size_info, "%s%s%s", SIZE_INFO_HEAD, send_total_size_str,
SIZE_INFO_TAIL);
                show(task->file_name, size_info, SHOW_DIRECTION_SYSTEM_INFO);
                free(size_info);

                send_wrap_split_data(talk_socket_fd, send_total_size_str, ETB);
                //recv head control ACK
                if (recv_equal_char(talk_socket_fd, ACK) == FALSE) return NULL;

                free_safe(send_total_size_str);
        }

        cinfo->friend_name = friend_name;
        cinfo->data_recv = data_recv;
        cinfo->connect_socket_fd = talk_socket_fd;
        cinfo->file_trans_fd = file_trans_fd;
        return cinfo;
}

void download_file(struct connect_info *cinfo)
{
        struct file_trans *task = find_file_trans_task(file_trans_control, cinfo->file_trans_fd);
        int result;
        unsigned char ch[1];
        if(task->connect_launcher == TRUE){//Launcher read send
                show(task->file_name, "send begin", SHOW_DIRECTION_SYSTEM_INFO);
                while (task->fin_size < task->total_size && !client_shutdown){
                        fread(ch, sizeof(unsigned char), 1, task->file_ptr);
                        result = send(cinfo->connect_socket_fd, ch, 1, 0);
                        if (result == -1) break;
                        task->fin_size += 1;
                }
                if (task->fin_size == task->total_size) {
                        show(task->file_name, "send finished",
SHOW_DIRECTION_SYSTEM_INFO);
                }

        }else{//Accepter recv append
                show(task->file_name, "download begin", SHOW_DIRECTION_SYSTEM_INFO);
                while (task->fin_size < task->total_size && !client_shutdown){
                        result = recv(cinfo->connect_socket_fd, ch, 1, 0);
                        if (result < 0 && !(errno == EINTR || errno == EWOULDBLOCK || errno ==
EAGAIN)) break;
                        task->fin_size += 1;
                        fwrite(ch, sizeof(unsigned char), 1, task->file_ptr);
                }
                if (task->fin_size == task->total_size) {
                        show(task->file_name, "download finished",
SHOW_DIRECTION_SYSTEM_INFO);
                }
```

```
	}

}

void destroy_download(struct connect_info *cinfo)
{
	destroy_file_trans(file_trans_control, cinfo->file_trans_fd);
	free_safe(cinfo);
}

Queue *init_split_data_recv()
{
	Queue *data_recv = (Queue *)malloc_safe(data_recv, sizeof(Queue));
	InitQueue(data_recv, sizeof(char *));
	return data_recv;
}

int recv_equal_char(socket_fd recv_socket_fd,char ch)
{
	char chstr[2] = {ch, 0};
	Queue *data_split = init_split_data_recv();
	int result = recv_unwrap_split_data(recv_socket_fd, data_split, NULL);
	if (result == TRUE) {
		char *data = init_data_recombine(data_split);
		recombine_data(data_split, data);
		result = strcmp(data, chstr) == TRUE?TRUE:FALSE;
		destroy_data_recombine(data);
	}
	destroy_split_data_recv(data_split);
	return result;
}

int recv_unwrap_split_data(socket_fd recv_socket_fd, Queue *data_recv, char *tail)
{
	int recv_result;
	int recv_end = 0;

	do {
		char *recvbuf = (char *)malloc_string_safe(recvbuf, RECV_BUFSIZE *
sizeof(char));
		recv_result = recv(recv_socket_fd, recvbuf, RECV_BUFSIZE - 1, 0);
		if (!compare_wrap(recvbuf, ETB)) {
			recv_end = 1;
			un_wrap(recvbuf, tail);
		}

		EnQueue(data_recv, &recvbuf);

		if (recv_result <= 0 && !(errno == EINTR || errno == EWOULDBLOCK || errno ==
EAGAIN))
			return FALSE;
	} while (!recv_end &&( recv_result > 0 || (recv_result < 0 && (errno == EINTR || errno ==
EWOULDBLOCK || errno == EAGAIN))));
	return TRUE;
}

void destroy_split_data_recv(Queue *data_recv)
{
	while (QueueLength(data_recv)) {//prevent malloc without free
		char *recvbuf;
		DeQueue(data_recv, &recvbuf);
		free_safe(recvbuf);
	}
	DestroyQueue(data_recv);
	free_safe(data_recv);
}

char *init_data_recombine(Queue *data_recv)
{
	int queue_length_max = QueueLength(data_recv);
	char *data = (char *)malloc_string_safe(data, RECV_BUFSIZE * sizeof(char) *
(queue_length_max + 1));
	return data;
}

void recombine_data(LinkQueue *data_recv,char *data)
{
	int queue_length_max = QueueLength(data_recv);
	int queue_length = queue_length_max;
	int data_length = 0;
	while (queue_length > 0) {
		char *recvbuf;
		DeQueue(data_recv, &recvbuf);
		if (data != NULL){
			memcpy(data + data_length, recvbuf, strlen(recvbuf));
			data_length += strlen(recvbuf);
		}
		free_safe(recvbuf);
		queue_length = QueueLength(data_recv);
	} //recombine all data to message
}

void destroy_data_recombine(char *data)
{
	free_safe(data);
}
#ifndef __talk_thread_H__
	#define __talk_thread_H__

	#include <pthread.h>
	#include <errno.h>
	#include <unistd.h>
	#include <stdlib.h>

	#include "Queue.h"
	#include "friend.h"
	#include "socket.h"
	#include "show_thread.h"
	#include "file_trans.h"



	extern Queue name_address;
	extern int client_shutdown;
	extern Queue connectors;
	extern Queue *file_trans_control;

	struct talk_thread_arg{
		socket_fd connect_socket_fd;
		int connect_launcher;//TRUE is launcher;FALSE is the accpeter
		int connect_type;//MESSAGE_CONNECT FILE_CONNECT
		int file_trans_fd;
	};


	struct connect_info{
		socket_fd connect_socket_fd;
		char *friend_name;
		Queue *data_recv;
		int file_trans_fd;
	};



	void *talk_thread(void *arg);
	int recv_equal_char(socket_fd recv_socket_fd,char ch);

	struct connect_info *init_message(socket_fd talk_socket_fd, char *friend_name, Queue
*data_recv);
	void show_message(struct connect_info *cinfo);
	void destroy_message(struct connect_info *cinfo);

	struct connect_info *init_download(socket_fd talk_socket_fd, char *friend_name, Queue
*data_recv, int file_trans_fd);
	void download_file(struct connect_info *cinfo);
	void destroy_download(struct connect_info *cinfo);

	Queue *init_split_data_recv();
	int recv_unwrap_split_data(socket_fd recv_socket_fd, Queue *data_recv, char
*tail);//FALSE when socket closed;TRUE when recv end
	void destroy_split_data_recv(Queue *data_recv);

	char *init_data_recombine(Queue *data_recv);
	void recombine_data(LinkQueue *data_recv,char *data);
```

```
        void destroy_data_recombine(char *data);
#endif /* __talk_thread_H__ */

#include "ulib.h"

int wrap(const char *from, const char tail, char *to)
{
        if(from == NULL || tail == '\0')
                return FALSE;
        int from_str_length = strlen(from) * sizeof(char);
        strncpy(to, from, from_str_length);
        memset(to + from_str_length, tail, 1);
        return TRUE;
}

int un_wrap(char *str, char *tail)
{
        if (str == NULL)
                return FALSE;
        size_t tail_position = (strlen(str) - 1) * sizeof(char);
        if (tail != NULL)
                *tail = *(str + tail_position);
        memset(str + tail_position, 0, 1);
        return TRUE;
}

int compare_wrap(const char *str,char tail)
{
        char tailstr[2] = {tail, 0};
        if (strcspn(str,tailstr) == strlen(str) - 1)
                return TRUE;
        else
                return FALSE;
}

char *long_to_string(long number)
{
        long _number = number;
        int size;
        for (size = 1; _number != 0; size += 1) {
                _number /= 10;
        }//end for

        char *str = (char *)malloc_string_safe(str, size);
        char ch;
        for (int i = size - 2; i != -1; i -= 1) {
                ch = number % 10;
                number /= 10;
                str[i] = ch + 48;
        }//end for
        return str;
}
```

```
#ifndef __ulib_H__
	#define __ulib_H__
	/*
	 * ulib.h
	 * This file is part of dp2pcsc
	 *
	 * Copyright (C) 2015 - Muromi Uhikari <chendianbuji@gmail.com>
	 *
	 * dp2pcsc is free software; you can redistribute it and/or
	 * modify it under the terms of the GNU Lesser General Public
	 * License as published by the Free Software Foundation; either
	 * version 2.1 of the License, or (at your option) any later version.
	 *
	 * dp2pcsc is distributed in the hope that it will be useful,
	 * but WITHOUT ANY WARRANTY; without even the implied warranty of
	 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
	 * Lesser General Public License for more details.
	 *
	 * You should have received a copy of the GNU Lesser General Public License
	 * along with dp2pcsc. If not, see <http://www.gnu.org/licenses/>.
	 */

	#include <stdlib.h>
	#include <string.h>


	#define malloc_safe(pointer, size) \
		malloc(size);\
		memset(pointer, 0, size)

	#define malloc_string_safe(pointer, size)\
		malloc(size + 1 * sizeof(char));\
		memset(pointer, 0, size + 1 * sizeof(char))

	#define free_safe(pointer) \
		if (pointer != NULL) free(pointer);\
		pointer = NULL;

	#ifndef TRUE
	#define TRUE 0
	#endif

	#ifndef	FALSE
	#define FALSE 1
	#endif

	#ifndef ERROR
	#define ERROR -1
	#endif

	int wrap(const char *from, const char tail,char *to);
	int un_wrap(char *str, char *tail);
```

```
        int compare_wrap(const char *str,char tail);
        char *long_to_string(long number);
#endif /* __ulib_H__ */
```

五.引用

[1] Om Malik (2013-08-11). "Zimmermann's Law: PGP inventor and Silent Circle co-founder Phil Zimmermann on the surveillance society — Tech News and Analysis". GigaOM. Retrieved 2013-08-20.