COMP4621 Programming Project Report

Name: CHAN, Chun Hin

Student ID: 20853893

Email: chchanec@connect.ust.hk

Table of Contents

client.c

i. send_return()

This function is used to search for a certain node in the linked node list start from the current node of the linked node list, and then send the RESPONSE message. Basically, I implement this function by considering 3 different situations.

The first situation is that if the current node passed from the input parameter is NULL, this means the current node and the linked node list is simply a null pointer and we should simply return NULL as it is obvious that we would not find the node with the correct file in this case.

The second situation is that if the current node is not NULL, meaning that the linked node list is not empty, we should start searching the linked node list to find the correct file. The two conditions that we should move on to the next node is when (a) the current node is not a null pointer (not NULL), and (b) the file map in the node does not match with the correct bitmap we want to find. Note that (b) can be implemented by using bitwise AND (&) to compare between the file map of the current node and the correct bitmap we want to find, if the bitmap file we would like to find is found, it gives a value not equal to 0; otherwise, if not match, it gives a result of 0.

The third situation is when we finish running the whole while loop in the second situation, and we found that current node is equal to NULL, this means that we have already transverse through the whole linked node list and reach the end to the linked node list. This indicates that the bitmap file we want to find does not found, so we can simply just return NULL.

ii. check_timeout()

This function is used to transverse through the whole context list to find all the data packets that are waiting for ACK and have already timeout. In case if there are data packets that are waiting for ACK and have already timeout, we should resend all of them. This can be done by using the if statements to do comparison. A data packet that are waiting for ACK is corresponds to the waiting_ack should have value of 1, which is the condition current->waiting_ack == 1. A data pack that has already timeout corresponds to the difference between the time now and the recorded clock time for last sending should be greater than or equal to the timeout value, which is the condition (now - current->clock >= TIMEOUT). Note that we should use logical AND (&&) to link the above two conditions together, resulting the condition current->waiting_ack == 1 && (now - current->clock >= TIMEOUT). If this condition is satisfied, we should update the recorded clock time for last sending this packet as the current time now, and after setting properly the IPv4 protocol, IP address and port number for the data members of clientaddr, we can use send_return() to resend the data packet with the corresponding current->file_idx, current->noack_node and current->noack_num.

iii. main()

This function is used for starting the server and handling different commands.

There are two parts in implementing this function, where both parts are related to handling the command UPDATE.

The first part is related to parsing the UPDATE input from the user. The implementation is quite trivial as suggested in the skeleton, I just follow the implementation in REGISTER, and I successfully parse the UPDATE input to obtain the destination IP address, the destination port number and the new bitmap file.

The second part is related to updating the bitmap file in the corresponding node and send ACK message back. First of all, we need to clean the information inside send_buf by filling the first MAXMSG bytes to be 0 using bzero(), which is to ensure the send_buf is clean before we pass the sequence number, a space blank and “ACK” to send_buf. The implementation of updating the bitmap file is trivial by just doing simple variable assignment. The implementation of preparing the ACK message and send it back is simple to implement. It can be implemented by mimicking the steps in implementing the parsing steps for UPDATE input from the user, such as using memcpy() and increment the send_idx appropriately. Finally, we can send the ACK message back by using sendto().

server.c

iv. rdt3_send()

This function is used to send and receive messages according to RDT3.0.

First, we reset parse_idx back to 0, and use bzero() to fill zeros to recv_buf, op and remain before we pass other things to these variables.

Then, we can use recvfrom() to get back the number of bytes I actually received (received_length) and the received message (recv_buf). There are two possible situations.

The first situation is when received_length is smaller than or equal to 0, this indicates there is an error when receiving the message back or we receive nothing. In this case, we should use sendto() to resend the message again, and also use set_timeout() to set the timeout for this message. After that, we skip to the next round of while loop.

The second situation is when received_length equals to 5, meaning that we receive an ACK packet. We can parse the message from recv_buf to get back the sequence number and the packet type. If we get the expected sequence number and the packet type is “ACK”, this means we received the right ACK message. In this case, we can set waiting to 0, meaning that we do not need to wait anymore.

v. send_update()

This function is used to send UPDATE messages to the server.

The implementation of this function is quite easy as I follow the suggestion by the instruction, I just take the implementation of send_register() as reference, and I implement the code to build a message packet for UPDATE type packet according to the description in the lab slides. I put the sequence number, a space blank, “UPDATE”, a space blank, IP address, port number and the file map in the appropriate position of the send buffer. At last, I use rdt3_send() to send the UPDATE type packet to server.

vi. p2p_server()

This function is used to help the peer-to-peer server to set up, and wait for the clients to connect to this server, and send file length and file content to the corresponding clients.

There are three parts that I need to implement here.

The first part is I need to initialize the pfds. To do so, I add server_fd to the pollfd set (pfds[0].fd = server_fd) and report that server_fd is ready to read on incoming connection (pfds[0].events = POLLIN). After that, I set fd_count to 1 as the pfds now contains server_fd.

The second part is I need to implement such that the server can receive a new connection from the clients. To do so, I use accept() to accept a new connection from a client. I also use new_socket to store the return value of accept(). If new_socket equals -1, this means we have some errors when accepting a new connection from the client. For error handling, I use perror() for error prompt and corresponding EXIT(EXIT_FAILURE) to handle this error if it happens. Otherwise, if there are no erros, I use add_to_pfds() to add the new file descriptor of the client socket to the pfds set.

The third part is to allow to send the file length and the file content to the peer-to-peer client. To do so, I first use send(new_socket, &file_size, sizeof(file_size), 0) to send the file length to the client. If send() returns a value less than 0, this means

there is errors when sending file length to the client. For error handling, I use perror() for error prompt and corresponding EXIT(EXIT_FAILURE) to handle this error if it happens.
Then, for the while loop, we would stay in the while loop until we reach the end of the file stream we are reading from. This is done by using the condition (feof(fp) == 0), as feof() would return non-zero value if we reach the end of the file stream, otherwise it would return 0 to indicate we have not yet reach the end of the file stream.
Inside the while loop, I use bzero() to clear buffer that is used for storing the file content that will be send. After that, I use fread(buffer, sizeof(char), MAXMSG, fp) to read the file content, and I use bytes_read to store number of bytes of file content have been read.
Next, I use ferror() to check whether there is error in the file stream. If there are errors, I use perror() for error prompt and corresponding EXIT(EXIT_FAILURE) to handle this error if it happens.
At last, I use send(new_socket, buffer, bytes_read, 0) to send the file content to the client. If send() returns a value less than 0, this means there is errors when sending file content to the client. For error handling, I use perror() for error prompt and corresponding EXIT(EXIT_FAILURE) to handle this error if it happens.

vii. p2p_client()

This function is used to help the peer to connect to the peer-to-peer server, and then the peer can receive the file from the server and save that received file.

Here is my code implementation description.

First, I use recv(sock, &file_len, sizeof(file_len), 0) to receive the file length of the received file. I have checked the C documentation of recv(), and if it returns a value smaller than 0, this means there is an error when we receive the file length, so I implement the error prompt using perror() and corresponding EXIT(EXIT_FAILURE) to handle this error if it happens.

Then, I initialize a variable no_bytes_received to count the number of bytes received.

Next, we will be inside the while loop as long as the number of bytes received is smaller than the file length of the received file. Inside the while loop, I use recv(sock, buffer, MAXMSG, 0) to receive the bytes from the received file and store it into buffer. I also use bytes_read to store the actual number of bytes received. For error handling, I use an if statement that if bytes_read is smaller than 0, this means that we have some error when receiving the bytes of the file, and we need to use perror() for error prompt and corresponding EXIT(EXIT_FAILURE) to handle this error if it happens. If there is no error, I use fwrite() to write the received bytes to the file stream we are writing to (fp). At the end of the while loop, we increment number of bytes received (no_bytes_received) by actual number of bytes received (bytes_read).

viii. main()

This function is used for creating the peer-to-peer server, and run different functions depending on the commands input by users.

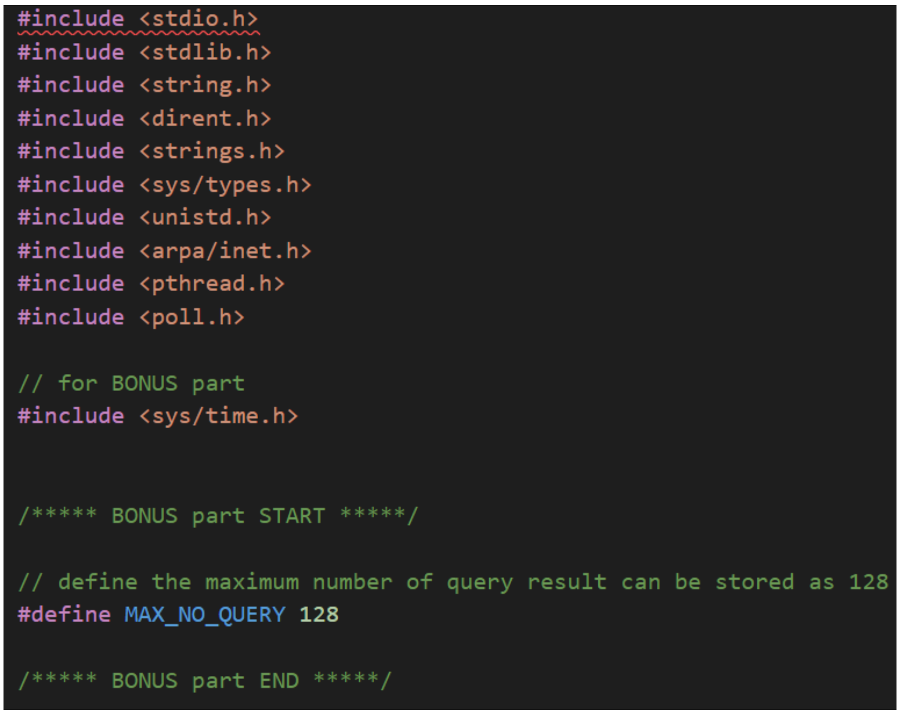
To start the peer-to-peer server using pthread, I use pthread_create(&tid, NULL, &p2p_server, (void *) &arg) to start it. For error handling, if there are any errors when creating the pthread, it would return an error number not equal to 0. In such situation, I use perror() for error prompting and corresponding EXIT(EXIT_FAILURE) to handle this error if it happens.

Bonus Part

Introduction

To implement the bonus part as stated in the lab slides, I only modified the client.c source code file. To be specific, I modified receive_query(), send_query() and main(). I have also defined a new constant MAX_NO_QUERY as 128, and import the header (#include <sys/time.h>). Let me talk about them in detail.

#include <sys/time.h>



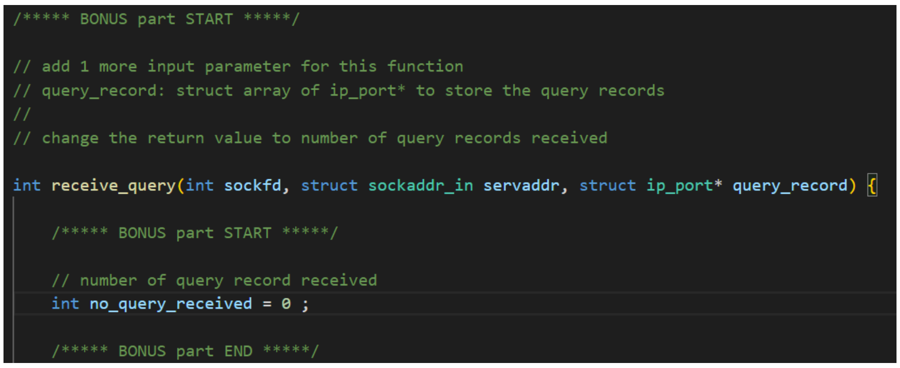
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <strings.h>
#include <sys/types.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <pthread.h>
#include <poll.h>

// for BONUS part
#include <sys/time.h>


/***** BONUS part START *****/

// define the maximum number of query result can be stored as 128
#define MAX_NO_QUERY 128

/***** BONUS part END *****/
```

I import this library as defines the timeval structure which includes the data members such as tv_sec and tv_usec. These data members would be used in client.c, so it is better to import this header.

MAX_NO_QUERY

I define MAX_NO_QUERY as 128, which is the maximum number of query result can be stored.

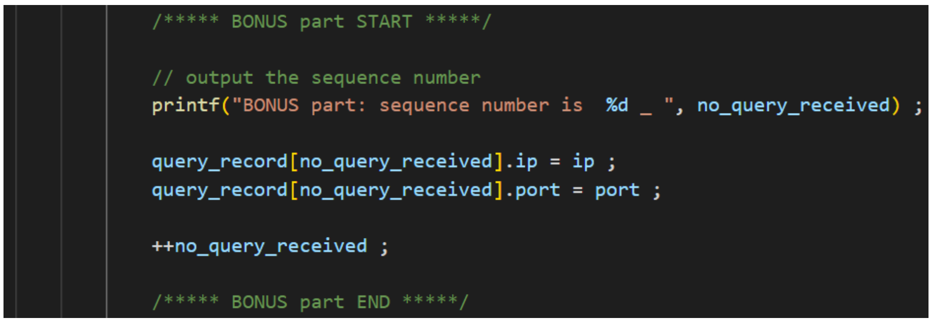
receive_query()



```
/***** BONUS part START *****/

// add 1 more input parameter for this function
// query_record: struct array of ip_port* to store the query records
//
// change the return value to number of query records received

int receive_query(int sockfd, struct sockaddr_in servaddr, struct ip_port* query_record) {

    /***** BONUS part START *****/

    // number of query record received
    int no_query_received = 0 ;

    /***** BONUS part END *****/
```

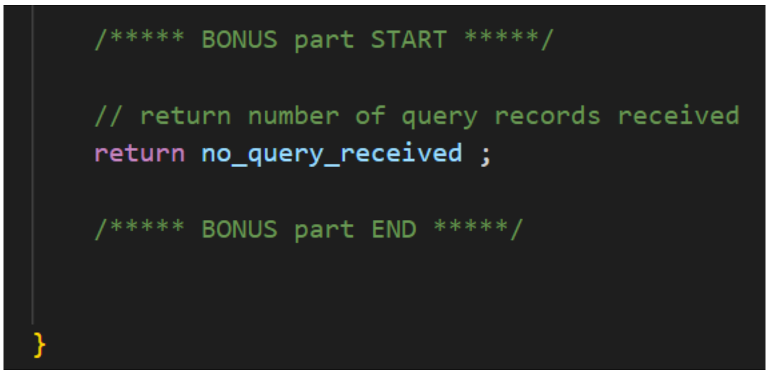
In this function, I add one more input parameter for this function, which is query_record that stores the query records.

I also initialize one variable no_query_received to count how many query records it received.



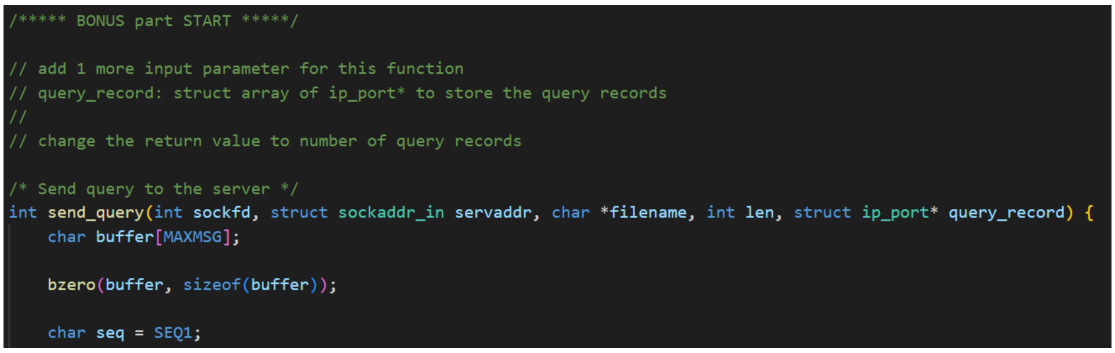
```
/***** BONUS part START *****/

// output the sequence number
printf("BONUS part: sequence number is  %d _ ", no_query_received) ;

query_record[no_query_received].ip = ip ;
query_record[no_query_received].port = port ;

++no_query_received ;

/***** BONUS part END *****/
```

Then, I changed the output of the query result by allowing that to show the sequence number as well, apart from the standard output that includes IP address and port number. Meanwhile, I use query_record to store the corresponding IP address and port number pair, and increment no_query_received by 1.

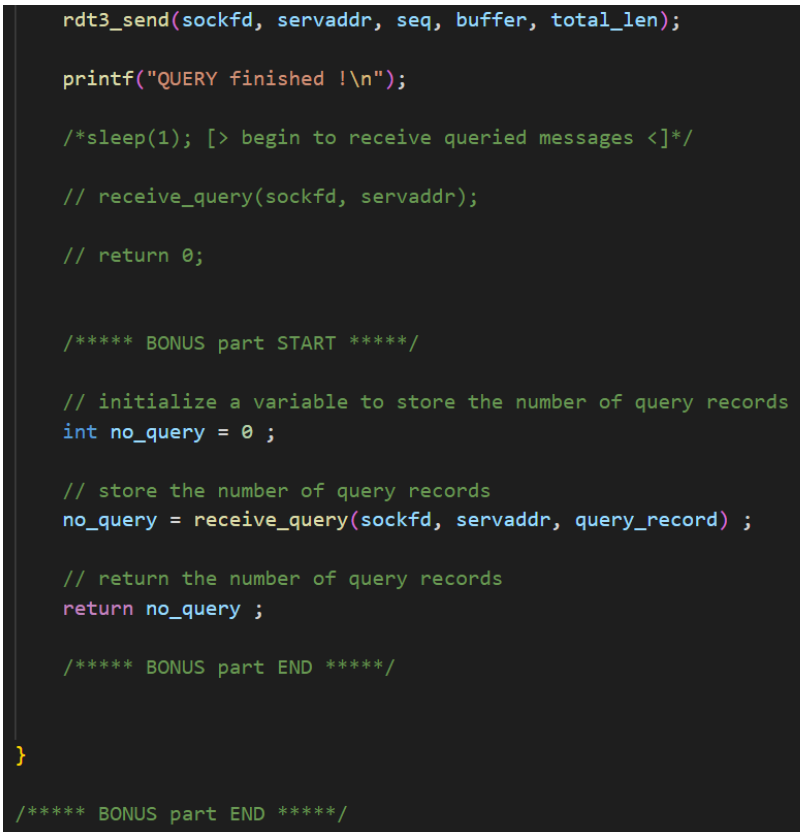

```
    /***** BONUS part START *****/

    // return number of query records received
    return no_query_received ;

    /***** BONUS part END *****/


}
```

When returning value, I change it to return the number of query records received.

send_query()


```
/***** BONUS part START *****/

// add 1 more input parameter for this function
// query_record: struct array of ip_port* to store the query records
//
// change the return value to number of query records

/* Send query to the server */
int send_query(int sockfd, struct sockaddr_in servaddr, char *filename, int len, struct ip_port* query_record) {
    char buffer[MAXMSG];

    bzero(buffer, sizeof(buffer));

    char seq = SEQ1;
```

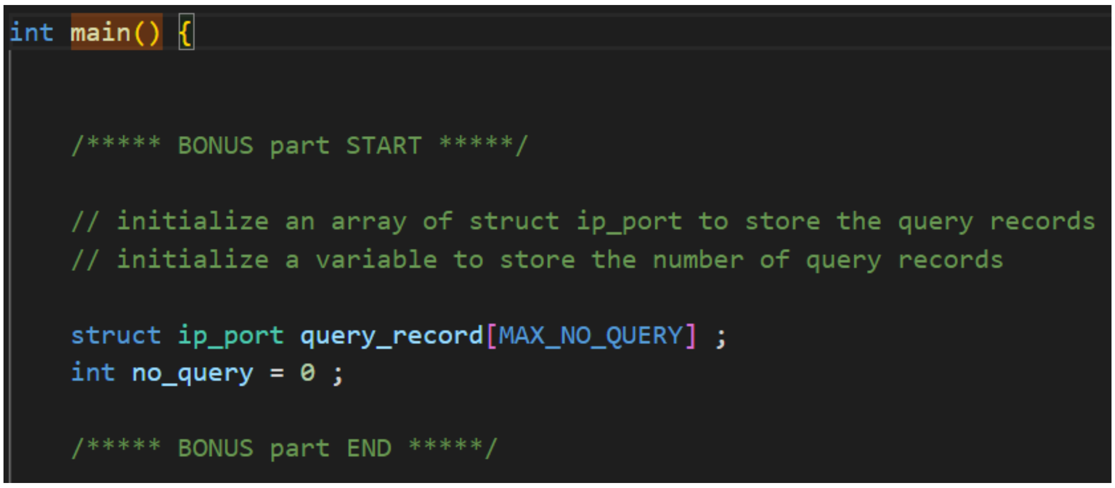
In this function, I add one more input parameter query_record that store the query records.



```
    rdt3_send(sockfd, servaddr, seq, buffer, total_len);

    printf("QUERY finished !\n");

    /*sleep(1); [> begin to receive queried messages <]*/

    // receive_query(sockfd, servaddr);

    // return 0;


    /***** BONUS part START *****/

    // initialize a variable to store the number of query records
    int no_query = 0 ;

    // store the number of query records
    no_query = receive_query(sockfd, servaddr, query_record) ;

    // return the number of query records
    return no_query ;

    /***** BONUS part END *****/


}

/***** BONUS part END *****/
```

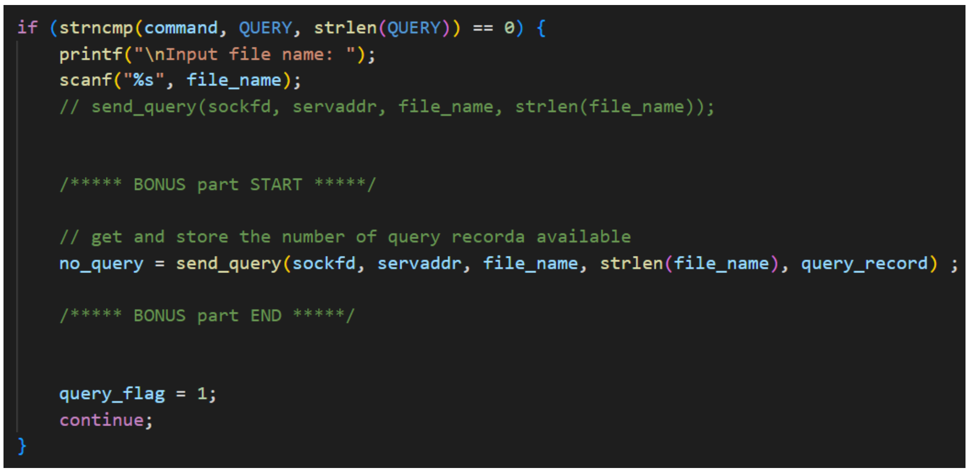
Then, near the end of this function, I initialize a variable no_query to count how many query records we receive after calling receive_query(). We return no_query as the return value.

main()



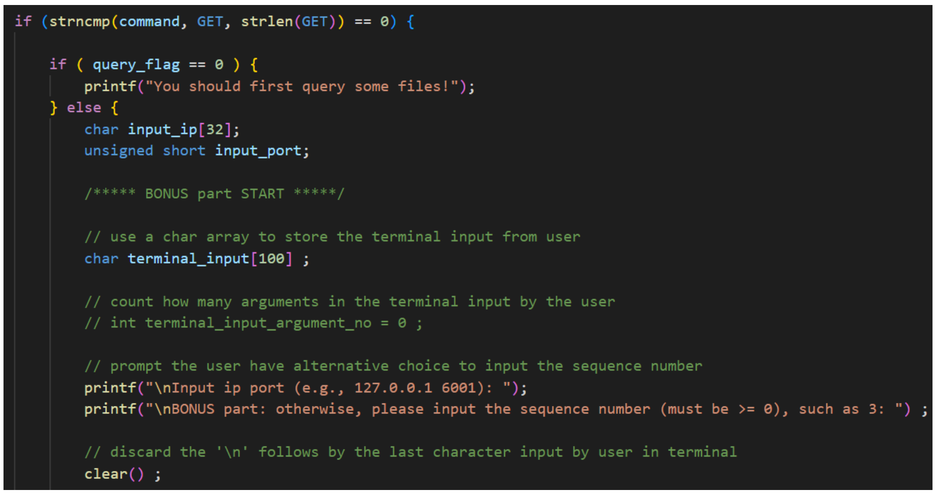
```
int main() {


    /***** BONUS part START *****/

    // initialize an array of struct ip_port to store the query records
    // initialize a variable to store the number of query records

    struct ip_port query_record[MAX_NO_QUERY] ;
    int no_query = 0 ;

    /***** BONUS part END *****/
```

In this function, I create 2 new variables. The first one is query_record, which is an array to store the query records. It can store MAX_NO_QUERY number of records at maximum. The second variable is no_query, which is to store the number of query records.



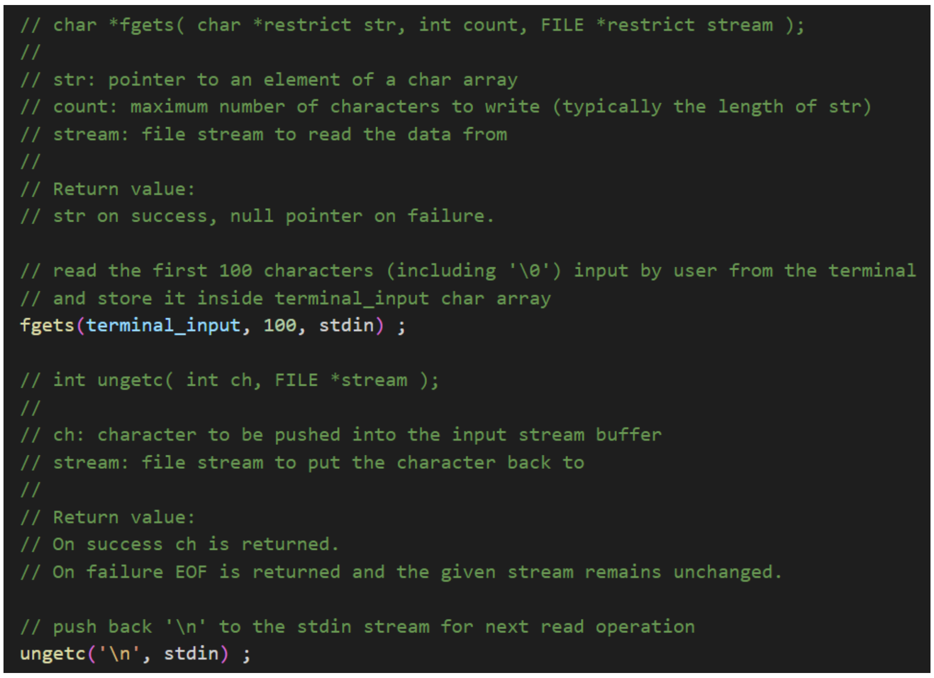
```
if (strncmp(command, QUERY, strlen(QUERY)) == 0) {
    printf("\nInput file name: ");
    scanf("%s", file_name);
    // send_query(sockfd, servaddr, file_name, strlen(file_name));


    /***** BONUS part START *****/

    // get and store the number of query recorda available
    no_query = send_query(sockfd, servaddr, file_name, strlen(file_name), query_record) ;

    /***** BONUS part END *****/


    query_flag = 1;
    continue;
}
```

If the user input the command “QUERY” we need to use no_query to store the number of query records after calling send_query().

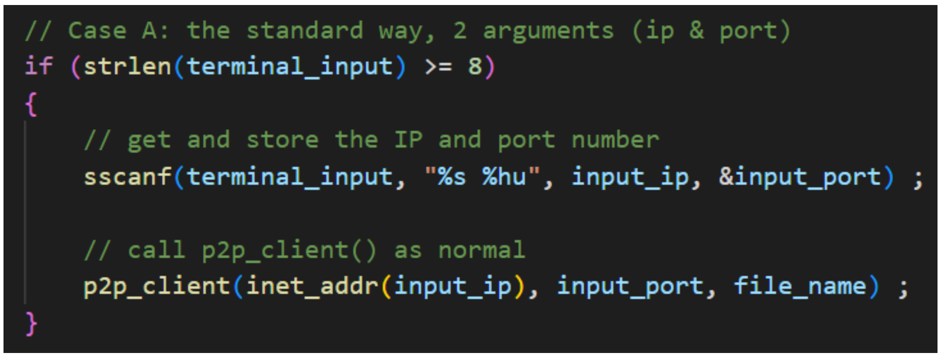


```
if (strncmp(command, GET, strlen(GET)) == 0) {

    if ( query_flag == 0 ) {
        printf("You should first query some files!");
    } else {
        char input_ip[32];
        unsigned short input_port;

        /***** BONUS part START *****/

        // use a char array to store the terminal input from user
        char terminal_input[100] ;

        // count how many arguments in the terminal input by the user
        // int terminal_input_argument_no = 0 ;

        // prompt the user have alternative choice to input the sequence number
        printf("\nInput ip port (e.g., 127.0.0.1 6001): ");
        printf("\nBONUS part: otherwise, please input the sequence number (must be >= 0), such as 3: ") ;

        // discard the '\n' follows by the last character input by user in terminal
        clear() ;
```

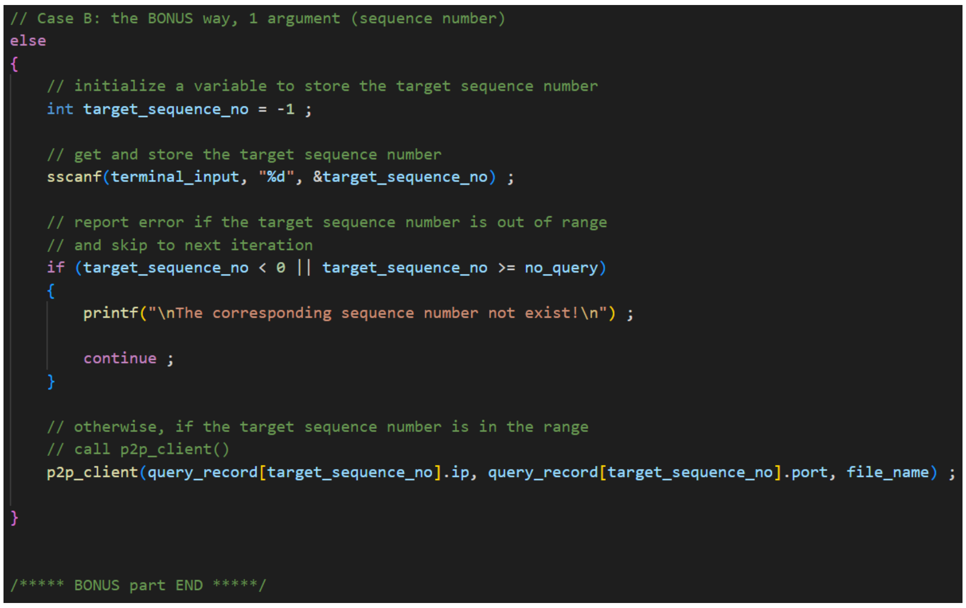
If the user input the command “GET”, we can prompt the user that he or she can input sequence number as well apart from the standard way. I also create another variable terminal_input to store the user input.



```c
// char *fgets( char *restrict str, int count, FILE *restrict stream );
//
// str: pointer to an element of a char array
// count: maximum number of characters to write (typically the length of str)
// stream: file stream to read the data from
//
// Return value:
// str on success, null pointer on failure.

// read the first 100 characters (including '\0') input by user from the terminal
// and store it inside terminal_input char array
fgets(terminal_input, 100, stdin) ;

// int ungetc( int ch, FILE *stream );
//
// ch: character to be pushed into the input stream buffer
// stream: file stream to put the character back to
//
// Return value:
// On success ch is returned.
// On failure EOF is returned and the given stream remains unchanged.

// push back '\n' to the stdin stream for next read operation
ungetc('\n', stdin) ;
```



```c
// Case A: the standard way, 2 arguments (ip & port)
if (strlen(terminal_input) >= 8)
{
    // get and store the IP and port number
    sscanf(terminal_input, "%s %hu", input_ip, &input_port) ;

    // call p2p_client() as normal
    p2p_client(inet_addr(input_ip), input_port, file_name) ;
}
```

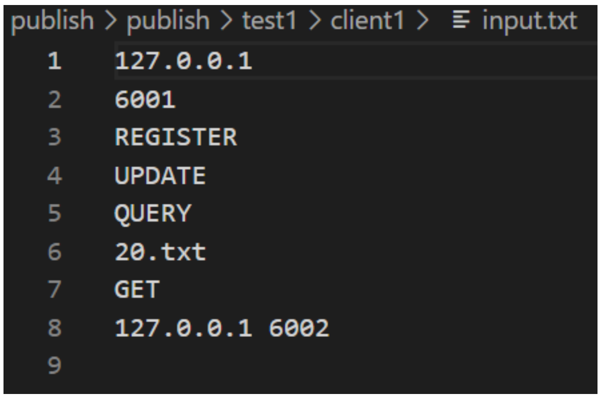


```c
// Case B: the BONUS way, 1 argument (sequence number)
else
{
    // initialize a variable to store the target sequence number
    int target_sequence_no = -1 ;

    // get and store the target sequence number
    sscanf(terminal_input, "%d", &target_sequence_no) ;

    // report error if the target sequence number is out of range
    // and skip to next iteration
    if (target_sequence_no < 0 || target_sequence_no >= no_query)
    {
        printf("\nThe corresponding sequence number not exist!\n") ;

        continue ;
    }

    // otherwise, if the target sequence number is in the range
    // call p2p_client()
    p2p_client(query_record[target_sequence_no].ip, query_record[target_sequence_no].port, file_name) ;

}


/***** BONUS part END *****/
```

I use fgets() and ungetc() to handle the user input carefully. We have 2 situations.

The first situation is if the user input is in the normal standard way, while the second situation is if the user input uses the method mentioned in the bonus description (i.e. the sequence number). This can be distinguished by whether the length of the input is of length 8 or above. If yes, the user is using the standard way, otherwise, the user is using the bonus way. No matter in which situation, we need to use sscanf() and p2p_client() to get back the correct input and initiate the client sockets respectively, just like in the original skeleton code.

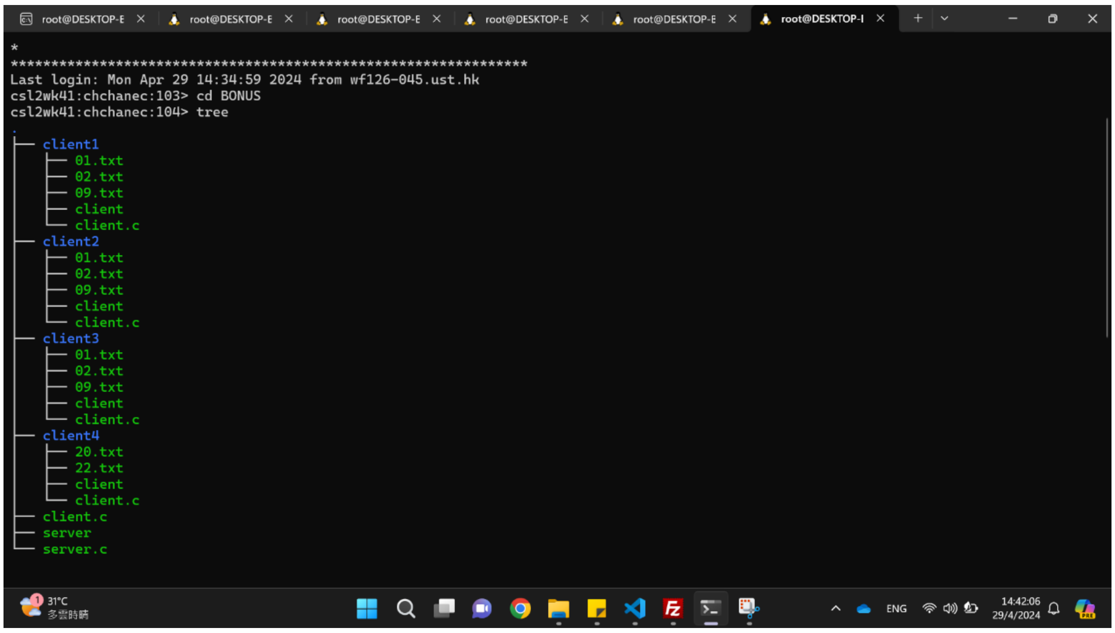
How to test the bonus



```
127.0.0.1
6001
REGISTER
UPDATE
QUERY
20.txt
GET
127.0.0.1 6002

```

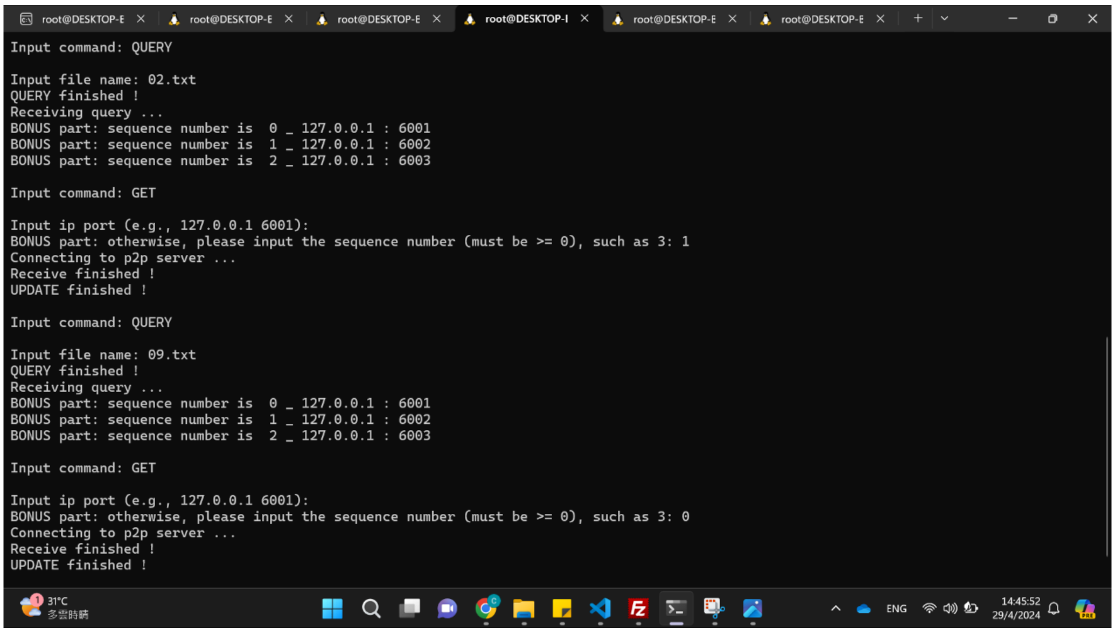
To test the bonus, it is very simple. In the test1, just change the last line of client1's input.txt (127.0.0.1 6002) to 0. This can still pass this test case.

Meanwhile, the following screenshots also proves the bonus part work:



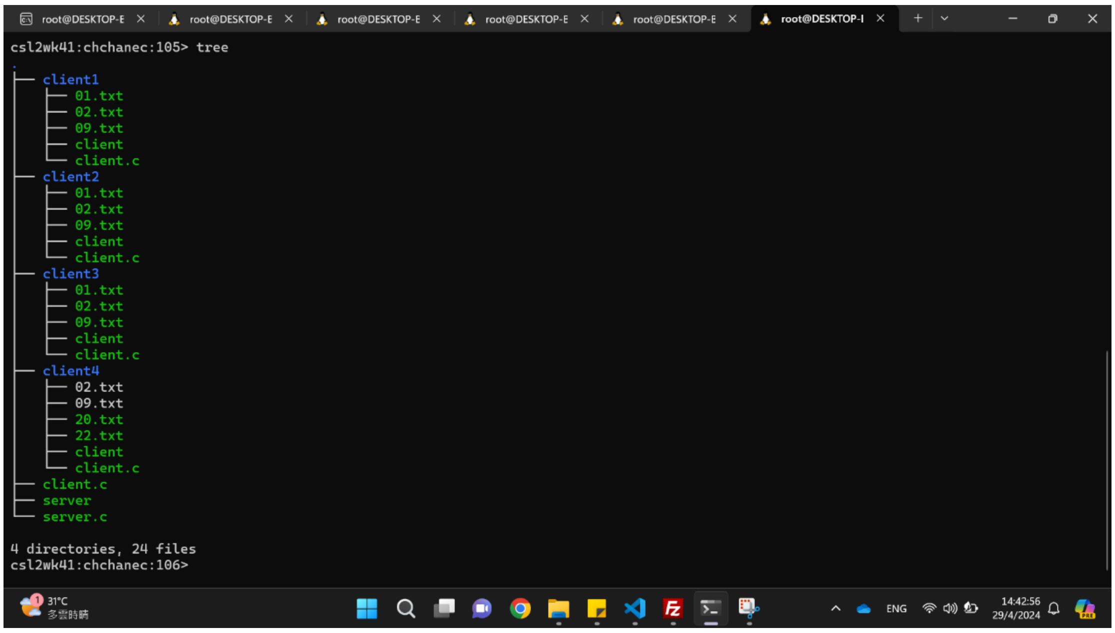
```
*
****************************************************************
Last login: Mon Apr 29 14:34:59 2024 from wf126-045.ust.hk
csl2wk41:chchanec:103> cd BONUS
csl2wk41:chchanec:104> tree
.
├── client1
│   ├── 01.txt
│   ├── 02.txt
│   ├── 09.txt
│   ├── client
│   └── client.c
├── client2
│   ├── 01.txt
│   ├── 02.txt
│   ├── 09.txt
│   ├── client
│   └── client.c
├── client3
│   ├── 01.txt
│   ├── 02.txt
│   ├── 09.txt
│   ├── client
│   └── client.c
├── client4
│   ├── 20.txt
│   ├── 22.txt
│   ├── client
│   └── client.c
├── client.c
├── server
└── server.c
```

Before the client4 request and get some files



```
Input command: QUERY

Input file name: 02.txt
QUERY finished !
Receiving query ...
BONUS part: sequence number is  0 _ 127.0.0.1 : 6001
BONUS part: sequence number is  1 _ 127.0.0.1 : 6002
BONUS part: sequence number is  2 _ 127.0.0.1 : 6003

Input command: GET

Input ip port (e.g., 127.0.0.1 6001):
BONUS part: otherwise, please input the sequence number (must be >= 0), such as 3: 1
Connecting to p2p server ...
Receive finished !
UPDATE finished !

Input command: QUERY

Input file name: 09.txt
QUERY finished !
Receiving query ...
BONUS part: sequence number is  0 _ 127.0.0.1 : 6001
BONUS part: sequence number is  1 _ 127.0.0.1 : 6002
BONUS part: sequence number is  2 _ 127.0.0.1 : 6003

Input command: GET

Input ip port (e.g., 127.0.0.1 6001):
BONUS part: otherwise, please input the sequence number (must be >= 0), such as 3: 0
Connecting to p2p server ...
Receive finished !
UPDATE finished !
```

Demonstrating the bonus feature works fine



```
csl2wk41:chchanec:105> tree
.
├── client1
│   ├── 01.txt
│   ├── 02.txt
│   ├── 09.txt
│   ├── client
│   └── client.c
├── client2
│   ├── 01.txt
│   ├── 02.txt
│   ├── 09.txt
│   ├── client
│   └── client.c
├── client3
│   ├── 01.txt
│   ├── 02.txt
│   ├── 09.txt
│   ├── client
│   └── client.c
├── client4
│   ├── 02.txt
│   ├── 09.txt
│   ├── 20.txt
│   ├── 22.txt
│   ├── client
│   └── client.c
├── client.c
├── server
└── server.c

4 directories, 24 files
csl2wk41:chchanec:106>
```

After the client4 request and get some files

Given Skeleton Part

This part would briefly describe the general structure of the given skeleton code part of both client.c and server.c, as instructed by the professor.

client.c

#include part

They are used to include the necessary header file so that we can use the functions and the structures that are related to socket programming in the following part of the code.

#define part

They are used to define some of the constants that are related to the tasks we need to implement in the following part of the code.

ip_port

This is a user-defined structure that can store the IP address and the port number.

set_timeout()

This is similar to we need to start the timer. This function uses setsockopt() to set the timeout for a given socket and a certain timeout value.

unset_timeout()

This is similar to we need to stop the timer. This function simply calls set_timeout() to stop the timer for a certain socket.

send_register()

This is used to send the REGISTER message along with the sequence number (seq), the IP address and port number to the server. The format of the REGISTER message follows that mentioned in the lab slide.

receive_query()

This is for the client used to receive the information about the available IP address and port number from the server, by parsing the RESPONSE message appropriately. After receiving the RESPONSE message, the client will send the ACK message to the server. A special case is that if the client received the FINISH message from the server, the client will just simply send the ACK message to the server.

send_query()

This is used to send the QUERY message along with the sequence number (seq) and the filename that we would like to request to the server. The format of the QUERY message follows that mentioned in the lab slide.

get_file_map()

This is used to prepare and construct a bitmap that represents which file(s) does the corresponding client has.

add_to_pfds()

This is used to add a new file descriptor to the set pfds, where it is an array of struct pointers. After adding a new file descriptor to the set, it will also properly assign the events data member of that array element just assigned with POLLIN to indicate that this socket is ready to read. It would then increment the count of file descriptor in the set by 1. Note that in the special case that, if we don't have enough room, it will double the size of pfds and allocate corresponding amount of memory.

del_from_pfds()

This is used to remove one certain file descriptor from the set, and follow by decrement the count of file descriptor in the set by 1. This is done by moving the last file descriptor in the set to the place where the file descriptor we would like to remove.

clear()

This is used to discard the '\n' follows by the last character input by user in terminal.

server.c

#include part

They are used to include the necessary header file so that we can use the functions and the structures that are related to socket programming in the following part of the code.

#define part

They are used to define some of the constants that are related to the tasks we need to implement in the following part of the code.

ip_port

This is a user-defined structure that can store the IP address and the port number.

node

This is a user-defined structure that in fact represents a linked node list. In each node, it contains the IP address, the port number, the bitmap for the files and the pointer to the next node.

rdt3_sender_ctx

This is a user-defined structure that in fact represents a linked list. Each node represents the context of an RDT3.0 sender. In each node, it contains the IP address, the port number, whether the sender is waiting for acknowledgment, the waiting acknowledgment number, the file index for QUERY and RESPONSE, the clock time that the last packet being sent, the waiting ack node, and the pointer to the next node in the linked node list.

get_current_time()

This is used to current time up to microsecond precision, and then return the current time expression in terms of milliseconds.

set_timeout()

This is similar to we need to start the timer. This function uses setsockopt() to set the timeout for a given socket and a certain timeout value.

unset_timeout()

This is similar to we need to stop the timer. This function simply calls set_timeout() to stop the timer for a certain socket.

init_node_list()

This is used to initialize an empty linked node list, so we simply return NULL.

insert_node()

This is used to insert a new node at the tail of the linked node list. The special case is that if the linked node list is empty, we set the head of the linked node list as the new node we want to insert.

query_node()

This is used to query a certain linked node list given a certain IP address and a certain port number. If it is found, we return that pointer to that node. Otherwise, we return NULL.

init_rdt3_sender_ctx_list()

This is used to initialize an empty linked context list of the RDT3.0 sender context. We return NULL to indicate the list is empty.

insert_ctx()

This is used to insert a new RDT3.0 sender context element at the tail of the linked context list. The special case is that if the linked context list is empty, we set the head of the linked context list as the new RDT3.0 sender context element we want to insert.

query_ctx()

This is used to query a certain RDT3.0 sender context given a certain IP address and a certain port number. If it is found, we return that pointer to that context. Otherwise, we return NULL.

send_finish()

This is used to send the FINISH message along with the sequence number (seq) to the client. The format of the FINISH message follows that mentioned in the lab slide.