CSE513 Project 1 Report

Peer-to-peer System with Centralized Index

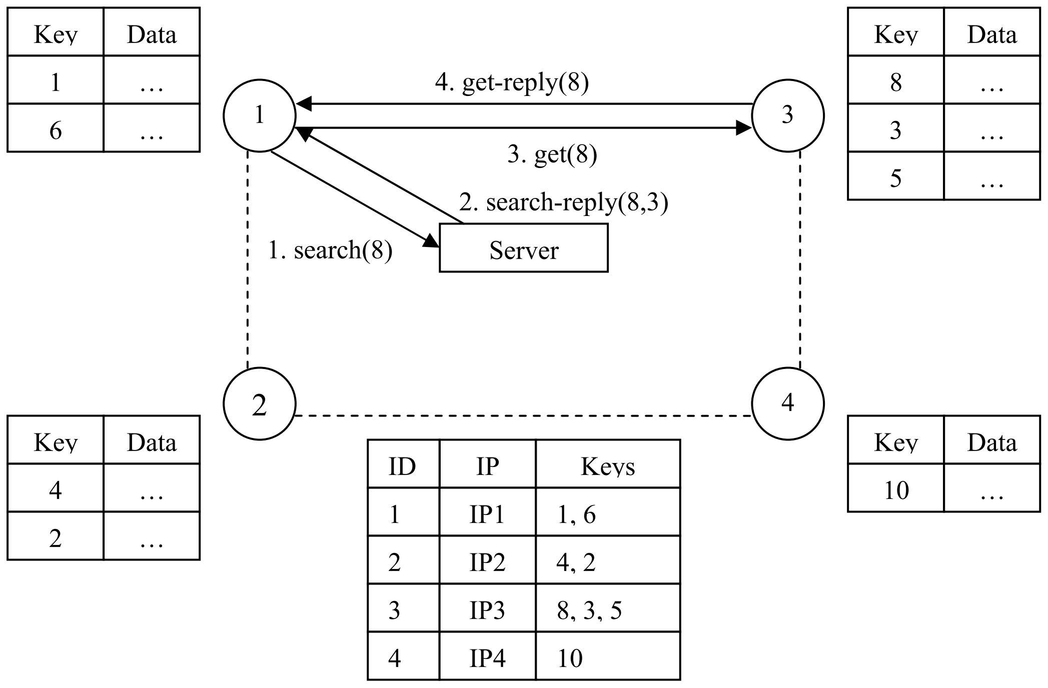
Abstract

As part of the CSE 513 class, the prototype of a peer-to-peer system has designed. In this project, a centralized server serves as the directory for locating the client that is hosting a particular data. Both of the client and server is multi-threaded in order to achieve high performance and simplicity.

Athichart Tangpong
Hendra Saputra

Table of Content

1. Overview



Figuer 1. Overiew of the system

In this project, there is a server responsible for maintaining the database of data location. When a new client want to join the system, it has to register itself with the server first. During the registration, the client publishes the keys of their data to the server. Whenever a client wants a data, it requests the location of the data from the server and then go get the data, as shown in Figure 1. The client 1 searches for the location of the data with the key of 8. The server returns its location which is the client 3. Then the client 1 requests the data from the client 3.

2.Protocol

The protocol is classified into two categories, namely client-to-server protocol and client-to-client protocol.

2.1) Client-to-Server Protocol

There are four kinds of message in this category, JOIN, LEAVE, DEAD and SEARCH, as shown in the Figure 2.

2	4	4	Number * sizeof(int)
JOIN	Node ID	Number	A list of keys

2	4
SEARCH	Key

2	4
LEAVE	Node ID

2	4
DEAD	Node ID

Figure 2. Summary of the client-to-client protocol

Whenever a client wants to register itself to the server, it submits a JOIN request, supplying its Node ID, number of keys maintained, and the keys themselves. In the case of search for a particular key, the client issues a SEARCH request to the server. The LEAVE request is for the situation that a client wants to leave the system. And if a client discovers the missing of another client, it reports the server with a DEAD message.

2.2) Client-to-Client Protocol

There is only one message in client-to-client protocol, namely GETDATA. This message is only related to the search message in client-to-server protocol. When the requesting client receives the return packet from the server in which contains the client host's node id and address, it will send the GETDATA message to the host client along with the key index.

There are two responds of this message. The first respond is the normal respond that contains the data itself. The second respond is the error respond. An error respond occurs whenever the host client is down and the server has not been notified yet. Whenever the requesting client receives this error message, it will notify the server that the host client is dead, so that the server can update its information.

3. Server

3.1) Structure

The server is equipped with the simplicity and performance of multi-threading technique. Upon the arrival of a new request, the server creates a new thread to service it. The thread's operation, described in more detail in the section 2.3, depends on the type of the request, extracted from the OPCODE field in the message.

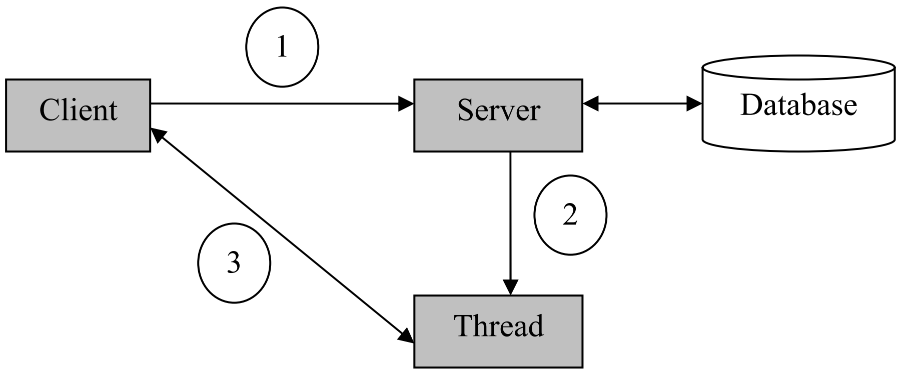


Figure 3. The server structure

Figure 3 illustrates the overview of the system. First of all, a client connects and submits a request to the server (1). Then the server creates a new thread to handle the request (2). This thread will consult the database, discussed in more detail in the next section. The thread interacts with the client through the newly established connection.

3.2) Database

The server maintains a database of clients and their keys. This database, shown in Figure 4, is nothing more than a linked-list equipped with a lock so as to guarantee mutual exclusion. Only one thread can access the database at any one time.

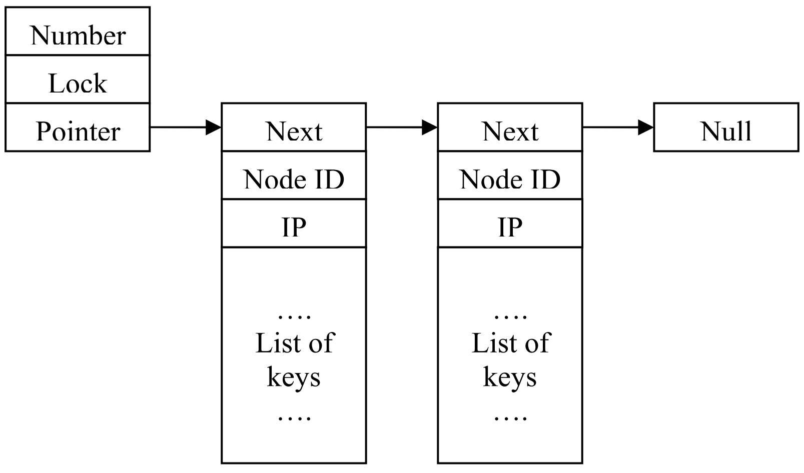


Figure 4. Server Database

Each element of the top-level linked-list is also a linked-list of keys residing on a particular client though illustrated with an array for the sake of comprehensibility. In this inner linked-list, the ID and IP address of a client is also maintained.

3.3) Protocol Handler

By consulting the OPCODE of a request, the newly created threads know how to response properly. For a JOIN request, the server threads receive the client NODE ID and their keys, and then insert them into the server database. For LEAVE and DEAD requests, the server threads delete the element of which the NODE ID field is identical to one in the request from the database. For a SEARCH request, the server threads search the entire database for the key requested by the client. The NODE ID and IP address of the client hosting the key is returned to the client if the search succeeds. Otherwise, a notification of search failure is returned instead.

4. Client

The structure of a client is similar to the structure of the server. Every client has a listening port that can receive any connection from other clients. Whenever a client receives a connection request on that specific port, it will create a new thread that can accept any request provided by the client-to-client protocol.

As explained in the previous chapter, there is only one message provided by the client-to-client protocol. When a client receives the GETDATA request, it will search its database (the client's database is similar to the server's database, except that the client's database contains the data and the key). Then the result of the search will be sent to the requesting client using the same connection.

5. Example

<u>When 2 clients connect to the server:</u>

Server:

```
ferengi 6% svc -p 6500
Node 10 (130.203.30.64) joined and published 8 keys.
keys: 1 2 3 4 5 6 7 8
Numbers of Clients = 1
Node 20 (130.203.18.32) joined and published 8 keys.
keys: 9 10 11 12 13 14 15 16
Numbers of Clients = 2
```

<u>When client with node id: 20 wants to search a key: 1</u>

Server:

```
Node 20 searched for the key 1 and found on the client 10.
```

Client 20:

```
Command shell [s/x] : s
Search for: 1
15 is in the client 10 (130.203.30.64)
>>>>> Trying to connect 130.203.30.64 to request my data .....
Title : Unchain my heart
ARTIST : Joe Cocker
COUNTRY : USA
COMPANY : EMI
YEAR : 1987
```

Client 10:

```
Command shell [s/x] :
>>>>> Incoming request for a key: 1
Title = Unchain my heart
ARTIST = Joe Cocker
COUNTRY = USA
```

```
COMPANY = EMI
YEAR = 1987

>>>>> The requested packet has been sent .......
```

<u>When client with node id 10 wants to search a key: 16</u>

Server:

```
Node 10 searched for the key 16 and found on the client 20.
```

Client 10:

```
Command shell [s/x] : s
Search for: 16
15 is in the client 20 (130.203.18.32)
>>>>> Trying to connect 130.203.18.32 to request my data .....
Title : A Big Willie style
ARTIST : A Will Smith
COUNTRY : A USA
COMPANY : A Columbia
YEAR : 1997
```

Client 20:

```
Command shell [s/x] :
>>>>> Incoming request for a key: 16
Title = A Big Willie style
ARTIST = A Will Smith
COUNTRY = A USA
COMPANY = A Columbia
YEAR = 1997

>>>>> The requested packet has been sent .......
```

<u>When client with node id 10 wants to search a key: 20 (no such key)</u>

Client 10:

```
Command shell [s/x] : s
Search for: 20
The key cant be found .....
Command shell [s/x] : s = search ; x = leave
Command shell [s/x] :
```

<u>When client with node id 10 is down, and yet notify the server while the client 20 needs to get data from node 10:</u>

Server:

```
Node 20 searched for the key 2 and found on the client 10.
Node 10 left.
Number of client = 1
```

Client 20:

```
Command shell [s/x] : s
Search for: 2
15 is in the client 10 (130.203.30.64)
[tcp_open]: unable to connect to server
>>>>> Trying to connect 130.203.30.64 to request my data .....
>>>>> Failed to connect the host machine .....
>>>>> Tell the server the host machine is down ....
Command shell [s/x] : s = search ; x = leave
Command shell [s/x] :
```

6. Comment

The system is fully operational with a minor limitation. In this version, a new thread is created upon the arrival of a new request. Unfortunately, the PTHREAD implementation allows only 2039 thread creations. Then the server can accept only

2039 connections. This nuisance can be eliminated easily by having the server adopt the thread-pool model, rather than on-demand thread creation.

7. Source Code

7.1) Summary

All the source code in this project, as well as its description, are shown in the Table 1.

Filename	Description
config.h	The header file for the protocol
fifo.h	The header file for the FIFO implementation
scanner.h	The header file for the scanner
tcp.h	The header file for the socket facilities
client.c	The implementation of the client
fifo.c	The implementation of the FIFO data structure
scanner.c	The scanner by which a client parses its configuration file and see what data it is hosting.
scanner.l	The lex source code for generating the client scanner (lex.yy.c)
server.c	The implementation of the server
tcp.c	The implementation of TCP socket facilities.
Makefile	Makefile for compiling the project

Table 1. Summary of the source code

7.2) client.c

```
/*
 * client.c : P2P client
 */
#include "config.h"
#include "fifo.h"
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/types.h>
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define PORT 6001
```

```
#define LISTENING_PORT 6022
#define GETDATA 10

typedef struct search_result_t search_result_t;
struct search_result_t {
        int node_id;
        char *hostname;
};


// Prototype
int read_data2(char *filename, fifo_t *key_list);
void printbuffer(int *ptr, int count);
void usage(char *command);
void CB_marshal_key(fifo_queue_t *qq, void *arg);
int CB_search_local_db(fifo_queue_t *qq, void *key);
keys_t *search_local_db(int key, fifo_t *key_list);
int join(char *machine_name, int port, int node_id, fifo_t *key_list);
search_result_t *search(char *machine_name, int port, int node_id, int key);
int leave(char *machine_name, int port, int node_id);
int dead(char *machine_name, int port, int node_id);
int leave_or_dead (short opcode, char *machine_name, int port, int node_id);
void *thr_func(void *ptr);
int perform(int fd);
int get_data(int fd);
int listening_post();
void create_input_window_thread();
void *input_window_thread(void *ptr);
int request_data(char* m_name, int nodeid, int key);

// Global variables
char *machine_name;
int server_port;
int node_id;
fifo_t *key_list;

int main(int argc, char **argv) {
        char opt, *delim="d:p:v?";
        int i;

        server_port=PORT;
        while((opt = getopt(argc,argv,delim)) != EOF) {
                switch(opt) {
                        case 'd' : node_id = atoi(optarg); break;
                        case 'p' : server_port = atoi(optarg); break;
                        case '?' :
                        default  : usage(argv[0]); exit(0);
                }
        }
```

```
	if( (argc-optind)!=2 ) {
		usage (argv[0]);
		exit(1);
	}
	machine_name = argv[optind];

	key_list = FIFO_create();

	read_data(argv[optind+1], key_list);

	// Demonstrate how to transmit data via the protocol
	join(machine_name, server_port, node_id, key_list);


	listening_post();
}

int listening_post() {
	struct	sockaddr_in cli_addr, serv_addr;
	int		sockfd, newsockfd;
	int		status, port, clilen;
	fd_set	watchset;
	struct	timeval tv;

	port = LISTENING_PORT;

	// Open a TCP socket (an Internet stream socket.)
	if( (sockfd=socket(AF_INET,SOCK_STREAM,0))<0 ) {
		perror("Server-Client:can't open stream socket\n");
		exit(1);
	}

	// Bind our local address so that the client can send to us.
	bzero((char *) &serv_addr,sizeof(serv_addr));
	serv_addr.sin_family	= AF_INET;
	serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
	serv_addr.sin_port		= htons(port);
	if( bind(sockfd,(struct sockaddr *) &serv_addr,
				sizeof(serv_addr))<0) {
		perror("Server-Client: can't bind local address\n");
		exit(1);
	}
	listen(sockfd,5);

	FD_ZERO(&watchset);
	FD_SET(sockfd, &watchset);
	tv.tv_sec = 10;
	tv.tv_usec = 0;

	create_input_window_thread();
```

```
        for(;;) {
                pthread_t *th;
                int *para;

                if(select(sockfd+1, &watchset, NULL, NULL, &tv) < 0) {
                        perror("Server: select error\n");
                        return -1;
                }

                if(FD_ISSET(sockfd, &watchset) != 0) {
                        clilen=sizeof(cli_addr);
                        if((newsockfd = accept(sockfd, (struct sockaddr *)
                                                &cli_addr,&clilen)) < 0) {
                                perror("Server: accept error\n");
                                return -1;
                        }

                        th = (pthread_t *)malloc(sizeof(pthread_t));
                        para = (int *)malloc(sizeof(int));
                        *para = newsockfd;

                        pthread_create(th, NULL, thr_func, (void *)para);
                } else {
                        FD_SET(sockfd, &watchset);
                }
        }
}

void *thr_func(void *ptr) {
        int sockfd = *((int *) ptr);

        perform(sockfd);
        close(sockfd);
        free(ptr);
        return 0;
}

int perform(int fd) {
        short opcode;

        if (readn(fd, &opcode, sizeof(short))<0)     {
                perror("Client: readn (at function perform) error\n");
                return -1;
        }
        switch(opcode) {
                case GETDATA: get_data(fd);
        }
}
```

```
int send_data(int fd, char *value) {
        int *sz;
        char *data;

        sz = (int* ) malloc (sizeof(int));
        *sz = strlen(value);
        data = (char *) malloc ((*sz) *sizeof(char));
        strncpy(data,value,*sz);

        if (writen(fd, sz, sizeof(int))<0 ) {
                perror ("client: writen error \n");
                return(-1);
        }

        if (writen(fd, data , (*sz)*sizeof (char))<0  ) {
                perror("Client: writen error \n");
                return (-1);
        }
}

int get_data(int fd) {
        char *data;
        int *sz;
        keys_t *object;
        int key_no;

        if (readn(fd, &key_no, sizeof(int))<0) {
                perror("Client: readn (at function get_data) error\n");
                return -1;
        }
        printf("\n>>>>> Incoming request for a key: %d\n",key_no);


        // Search local DB
        if((object = search_local_db(key_no, key_list)) != NULL) {
                printf("Title = %s\n", object->title);
                printf("ARTIST = %s\n", object->artist);
                printf("COUNTRY = %s\n", object->country);
                printf("COMPANY = %s\n", object->company);
                printf("YEAR = %s\n", object->year);
                printf("\n");

                send_data(fd,object->title);
                send_data(fd,object->artist);
                send_data(fd,object->country);
                send_data(fd,object->company);
                send_data(fd,object->year);

                printf(">>>>> The requested packet has been sent ....... \nCommand
        shell [s/x] : ");
        }
}

void create_input_window_thread() {
        pthread_t *th;
        int *para;

        para = (int *) malloc (sizeof(int));
        th = (pthread_t *) malloc(sizeof(pthread_t));
        pthread_create(th,NULL,input_window_thread,(void *) para);
}

void *input_window_thread(void *ptr) {
        char command[3];
        int wantedkey;
        search_result_t *result;

        for (;;)    {
                printf("Command shell [s/x] : ");
                scanf("%c",command);

                switch(command[0]) {
                case 's':
                        printf("Search for: ");
                         scanf("%d",&wantedkey);
                         if( (result = search(machine_name, server_port, node_id,
        wantedkey)) != NULL) {
                                printf("15 is in the client %d (%s)\n", result->node_id,
                                result->hostname);
                           request_data(result->hostname, result->node_id,
        wantedkey);
                            } else
                                printf("The key cant be found ..... \n");
                        break;
                case 'x':
                         leave(machine_name, server_port, node_id);
                         printf("Thank you for using our program ..... \n");
                         printf("Please dont make any segmentation fault !!!!, >:( \n");
                         exit(0);
                default:
                         printf("s = search ; x = leave \n");
                }
        }
        return 0;
}

int receive_data(int sockfd, char* note) {
        int sz;
        char *data;
        if ( readn(sockfd, &sz, sizeof(int))<0) {
              perror("Client: readn error \n");
        }
        data = (char *) malloc (sz * sizeof(char));
        if ( readn(sockfd, data, sz * sizeof(char))<0) {
              perror("Client: read error - search \n");
                 return(-1);
        }
        data[sz]='\0';
        printf("%s : %s \n",note,data);
}

int request_data(char* m_name, int nodeid, int key) {
        short opcode = GETDATA;
        int sockfd = tcp_open(m_name, LISTENING_PORT);
        int key_no = key;

        printf(">>>>> Trying to connect %s to request my data ..... \n",m_name);

        if (sockfd<0)  {
                printf(">>>>> Failed to connect the host machine ..... \n");
                printf(">>>>> Tell the server the host machine is down .... \n");
                dead(machine_name, server_port, nodeid);
                return(-1);
          }

        // Send opcode
        if( writen(sockfd, &opcode, sizeof(short))<0 ) {
                perror("Client: writen error\n");
                return(-1);
        }

         // Send the key
        if( writen(sockfd, &key_no, sizeof(int))<0 ) {
                perror("Client: writen error\n");
                return(-1);
        }

        receive_data(sockfd,"Title");
        receive_data(sockfd,"ARTIST");
        receive_data(sockfd,"COUNTRY");
        receive_data(sockfd,"COMPANY");
        receive_data(sockfd,"YEAR");
        close(sockfd);
}


// --------------------------------------- Sub-routines -----------------------------------------
/*
 * read_data2
 * Functionality : Read the list of keys and associated data, then
 *                 keep them in the key_list. For testing only. (Deprecated)
 * Return Value :  1 if successful, -1 otherwise.
 * Parameter :
 *         filename - filename
 *         key_list - the FIFO_queue to keep the keys and their associated data
 */
int read_data2(char *filename, fifo_t *key_list) {
          int  count, tmp, *ptr;
          FILE *fp;
          char buffer[1024], data[1024];

          if( (fp=fopen(filename,"r")) == NULL) {
                    puts("Cannot open file\n");
                    return -1;
          }

          count = 0;
          while( fscanf(fp,"%d", &tmp) != EOF) {
                    ptr = (int *)malloc(sizeof(int));
                    *ptr = tmp;
//                  printf("%d ", tmp);
                    FIFO_enqueue(key_list, FIFO_element(ptr));
                    count++;
          }
//        printf("\n");

          fclose(fp);
          return(count);
}

/*
 * usage
 * Functionality : Print help message
 * Return Value : NONE
 * Parameter :
 *         command - command
 */
void usage(char *command) {
          printf("usage : %s <server> <input file>\n", command);
          printf("Valid option is\n");
          printf(" p <port> : connect server at the specified port\n");
          printf(" v        : print version of %s\n", command);
          printf(" ?        : print this help message\n");
}

/*
 * CB_marshal_key
 * Functionality : Arrange the keys in the key_list so as to be able
 *                 to transfer to the server during joining
 * Return Value : NONE
 * Parameter :
 *        qq  - an element in the key_list table
 *        arg - the pointer to the position to be written to in an array
 */
void CB_marshal_key(fifo_queue_t *qq, void *arg) {
          int *pointer = (int *)*((int *)arg);
          keys_t *value;

          if(qq->ptr != NULL) {
                    value = (keys_t *)qq->ptr;
                    *pointer = value->key;
                    *((int *)arg) += 4;
          }
}

int CB_search_local_db(fifo_queue_t *qq, void *key) {
          keys_t *object;

          if(qq->ptr != NULL) {
                    object = (keys_t *)qq->ptr;

                    if(object->key == *((int *)key))
                              return 1;
                    else
                              return 0;
          }
}

keys_t *search_local_db(int key, fifo_t *key_list) {
          fifo_queue_t *qq;

          if((qq = FIFO_search2(key_list, CB_search_local_db,
                                              (void *)&key)) != NULL)
                    return (keys_t *)qq->ptr;
          else
                    return NULL;
}

/*
  * join
  * Functionality: register the client to the server and publish the keys
  *                of data that are hosted by the client.
  * Return Value: 1 if successful, otherwise -1
  * Parameter:
  *       machine_name - server name
  *       port         - server port
  *       node_id      - client's node ID
  *       key_list     -
  */
```

```
int join(char *machine_name, int port, int node_id, fifo_t *key_list) {
        short opcode = JOIN;
        int sockfd = tcp_open(machine_name, port);
        int number,*buffer, *ptr;

        number = FIFO_count(key_list);

        // Send opcode
        if( writen(sockfd, &opcode, sizeof(short))<0 ) {
                printf("Client: writen error\n");
                return(-1);
        }

        // Send node_id
        if( writen(sockfd, &node_id, sizeof(int))<0 ) {
                printf("Client: writen error\n");
                return(-1);
        }

        // Send the number of keys
        if( writen(sockfd, &number, sizeof(int))<0 ) {
                printf("Client: writen error\n");
                return(-1);
        }

        // Send the list of keys
        buffer = (int *)malloc(number*sizeof(int));
        ptr = buffer;
        FIFO_foreach2(key_list, CB_marshal_key, (void *)&ptr);

#ifdef __DEBUG__
        { int i, *buffer2;
                buffer2 = buffer;
                for(i=0; i<number; i++)
                        printf("%d ", *(buffer2)++);
                printf("\n");
        }
#endif

        if( writen(sockfd, buffer, number*sizeof(int))<0 ) {
                printf("Client: writen error\n");
                return(-1);
        }

        close(sockfd);
        return(0);
}

/*
 * search
 * Functionality: search for a data with associated key
 * Return Value: Node Id and the hostname of the client hosting the data if
 *               found, otherwise NULL
 * Parameter:
 *       machine_name - server name
 *       port         - server port
 *       node_id      - client's node ID
 *       key          - the key to be searched for
 */
search_result_t * search(char *machine_name, int port, int node_id, int key) {
        short opcode = SEARCH;
        int sockfd = tcp_open(machine_name, port);
        int result;

        // Send opcode
        if( writen(sockfd, &opcode, sizeof(short))<0 ) {
                printf("Client: writen error\n");
                return NULL;
        }

        // Send node_id
        if( writen(sockfd, &node_id, sizeof(int))<0 ) {
                printf("Client: writen error\n");
                return NULL;
        }

        // Send the key
        if( writen(sockfd, &key, sizeof(int))<0 ) {
                printf("Client: writen error\n");
                return NULL;
        }

        // Read the result back from the server
        if( readn(sockfd, &result, sizeof(int))<0 ) {
                printf("Client: writen error\n");
                return NULL;
        }

        // Read the hostname
        if(result != -1) {
                search_result_t *res;
                int size;

                res = (search_result_t *)malloc(sizeof(search_result_t));
                res->node_id = result;

                // Read the size
                if( readn(sockfd, &size, sizeof(int))<0 ) {
                        printf("Client: writen error\n");
                        return NULL;
                }

                // Read the hostname
                res->hostname = (char *)malloc(size);
                if( readn(sockfd, res->hostname, size)<0 ) {
                        printf("Client: writen error\n");
                        return NULL;
                }

                return res;
        }

        return NULL;
}

/*
 * leave
 * Functionality : Send a LEAVE message
 * Return Value : 1 if successful, otherwise -1
 * Parameter :
 *      machine_name        - server name
 *      port                        - server port
 *      node_id                     - client's node ID
 */
int leave(char *machine_name, int port, int node_id) {
        return leave_or_dead(LEAVE, machine_name, port, node_id);
}

/*
 * dead
 * Functionality : Send a DEAD message to notify the server of the
 *      death of a client
 * Return Value : 1 if successful, otherwise -1
 * Parameter :
 *      machine_name  - server name
 *      port          - server port
 *      node_id       - client's node ID
 */
int dead(char *machine_name, int port, int node_id) {
        return leave_or_dead(DEAD, machine_name, port, node_id);
}

/*
 * leave_or_dead
 * Functionality : unregister the client from the server
 * Return Value : 1 if successful, otherwise -1
 * Parameter :
 *      opcode       - operation (DEAD or LEAVE);
 *      machine_name - server name
 *      port         - server port
 *        node_id            - client's node ID
 */
int leave_or_dead (short opcode, char *machine_name, int port, int node_id) {
          int sockfd = tcp_open(machine_name, port);
          int result;

          // Send opcode
          if( writen(sockfd, &opcode, sizeof(short))<0 ) {
                    printf("Client: writen error\n");
                    return(-1);
          }

          // Send node_id
          if( writen(sockfd, &node_id, sizeof(int))<0 ) {
                    printf("Client: writen error\n");
                    return(-1);
          }

          return 0;
}
```

7.3) fifo.c

```
/*
 * fifo.c : FIFO with mutex lock
 * FIFO with mutex lock
 * $Id: fifo.c,v 1.2 2002/11/18 09:29:38 tangpong Exp $
 */
#include <stdio.h>

#include "fifo.h"

#ifdef __FIFO_TEST__
/*
 * Routines for testing FIFO
 */

const int MAX= 300;

void func1(fifo_queue_t *q) {
          printf("foreach: %d\n", *((int *)q->ptr));
}

int test(fifo_queue_t *q) {
          if( *((int *)q->ptr) == 3999 )
                    return 1;
          else
                    return 0;
}

int CB_test_remove2(fifo_queue_t *qq, void *arg) {
          if( *((int *) qq->ptr) == *((int *) arg) )
```

```
                return 1;
        else
                return 0;
}

void pause() {
        printf("Paused: Please press [RETURN] to continue\n");
        getchar();
}

int main(int argc, char **argv) {
        fifo_t *ff;
        fifo_queue_t *qq;
        int j, arg;

        ff = FIFO_create();

        // Test FIFO_enqueue
        for(j=0; j<=MAX; j++) {
                int *i;

                i = (int *)MALLOC(sizeof(int));
                *i = j;
                qq = FIFO_element(i);
                printf("Enqueue: %d\n", *((int *)qq->ptr));
                FIFO_enqueue(ff, qq);
        }

        pause();

        // Test FIFO_search
        if( (qq = FIFO_search(ff, test)) != NULL ) {
                printf("FOUND: %d\n", *((int *)qq->ptr));
        }

        pause();

        // Test FIFO_remove2
        for(j=100;j<=200; j++) {
                if( (qq = FIFO_remove2(ff, CB_test_remove2, &j)) != NULL ) {
                        printf("FOUND: %d\n", *((int *)qq->ptr));
                }
        }

        pause();

        // Test FIFO_foreach
        FIFO_foreach(ff, func1);

        pause();

/*
        // Test FIFO_dequeue
        while( (qq = FIFO_dequeue(ff)) != NULL ) {
                printf("Dequeue: %d\n", *((int *)qq->ptr));
                FREE(qq);
        }
*/

        FIFO_destroy(ff);
}

#endif

/*
 * MALLOC
 * Functionality: allocate memory (usign malloc), and check error
 * Return value: The address of the memory allocated on success;
 *              otherwise NULL
 * Parameter:
 *              size - the size of memory to be allocated
 */
void * MALLOC(int size) {
        void *temp;

        if( (temp = (void *)malloc(size)) == NULL) {
                fprintf(stderr, "[MALLOC] malloc error!\n");
                exit(-1);
        }

        return temp;
}

/*
 * FREE
 * Functionality: free allocated memory, if not NULL
 * Return value: NONE
 * Paramter:
 *      ptr - the pointer to be deallocated
 */
void FREE(void *ptr) {
        if(ptr != NULL)
                free(ptr);
}

/*
 * FIFO_create
 * Functioanlity: Create a new FIFO
 * Return value: a pointer to a FIFO
 * Parameter: NONE
 */
fifo_t * FIFO_create() {
        fifo_t *f;

        f = (fifo_t *)MALLOC(sizeof(fifo_t));
        f->count = 0;
#ifdef __PTHREAD__
        f->lock = (pthread_mutex_t *)MALLOC(sizeof(pthread_mutex_t));
        if(pthread_mutex_init(f->lock, NULL) != 0) {
                printf("Pthread Init failed\n");
                return NULL;
        }
#else
        f->lock  = (lwp_mutex_t *)MALLOC(sizeof(lwp_mutex_t));
#endif
        f->queue = NULL;

        return f;
}

/*
 * FIFO_destroy
 * Functioanlity: Free all memory allocated to a FIFO, including
 *                all of its elements
 * Return value: NONE
 * Parameter:
 *                f - a pointer to the FIFO
 */
void FIFO_destroy(fifo_t *f) {
        fifo_queue_t *cur, *free_me;

        if(f == NULL) return;

        FIFO_lock(f);

        /* Free all queue */
        cur = f->queue;
        f->queue = NULL;
        while(f->count > 0) {
                free_me = cur;
                cur = cur->next;
                FREE(free_me);
                f->count--;
        }

        FIFO_unlock(f);

        FREE(f->lock);          // Free mutex lock
}

/*
 * FIFO_element
 * Functioanlity: Encapsulate a void pointer in a fifo_queue_t in
 *                order to be eligible to be put into a FIFO
 * Return value: the fifo_queue_t of the input void pointer
 * Parameter:
 *                ptr -  anything that you want to be put into the FIFO
 */
fifo_queue_t * FIFO_element(void *ptr) {
        fifo_queue_t  *q;

        q = (fifo_queue_t *)MALLOC(sizeof(fifo_queue_t));
	q->ptr = ptr;
	q->next = NULL;

	return q;
}

/*
 * FIFO_is_empty
 * Functioanlity: Check if the queue is empty
 * Return value: 1 if empty; otherwise 0; -1 for error
 * Parameter:
 *		f - a pointer to the FIFO
 */
int FIFO_is_empty(fifo_t *f) {
	int temp;

	if(f == NULL) return -1;

	FIFO_lock(f);
	temp = f->count;
	FIFO_unlock(f);

	return temp == 0;
}

/*
 * FIFO_count
 * Functioanlity: Return the number of elements
 * Return value: Number of elements
 * Parameter:
 *		f - a pointer to the FIFO
 */
int FIFO_count(fifo_t *f) {
	int temp;

	if(f == NULL) return -1;

	FIFO_lock(f);
	temp = f->count;
	FIFO_unlock(f);

	return temp;
}

/*
 * FIFO_enqueue
 * Functioanlity: Put an element into the queue
 * Return value: NONE
 * Parameter:
 *		f - a pointer to the FIFO queue
 *		q - an element to put into the queue
 */
void FIFO_enqueue(fifo_t *f, fifo_queue_t *q) {
	fifo_queue_t *cur;
	if(f == NULL) return;

	FIFO_lock(f);

	if(f->queue == NULL) { /* f is empty */
		f->queue = q;
		f->count++;
	} else {					/* f is not empty */
		cur = f->queue;

		/* Move cur to the last element of the queue */
		while(cur->next != NULL) {
			cur = cur->next;
		}

		/* Append q */
		cur->next = q;
		f->count++;
	}

	FIFO_unlock(f);
}

/*
 * FIFO_dequeue
 * Functioanlity: Get and remove the element at the head of the queue
 * Return value: an element at the head of the queue; NULL in case that
 *		the FIFO is empty
 * Parameter:
 *		f - an pointer to the FIFO queue
 */
fifo_queue_t * FIFO_dequeue(fifo_t *f) {
	fifo_queue_t *cur;

	if(f == NULL) return NULL;

	FIFO_lock(f);

	cur = NULL;
	if(f->count != 0) {
		cur = f->queue;
		f->queue = f->queue->next;
		cur->next = NULL;		// For the sake of safety
		f->count--;
	}

	FIFO_unlock(f);

	return cur;
}

/*
 * FIFO_search
 * Functioanlity: Search the FIFO "q" for an element that match the
 *              requirement specified in the callback subroutine "func"
 * Return value: an element if found; otherwise NULL
 * Parameter:
 *              f - a pointer to FIFO
 *              func - a calback function "int func_name(fifo_queue_t *q)" If
 *                      the element matches, return 1; otherwise 0.
 */
fifo_queue_t * FIFO_search(fifo_t *f,  int (*func)(fifo_queue_t *)) {
        fifo_queue_t *cur, *dummy, *found;

        if(f == NULL) return NULL;

        FIFO_lock(f);

        found = NULL;  /* Not found */
        if(f->count != 0) {
                cur = f->queue;

                do {
                        dummy = cur;
                        cur = cur->next;
                        if(func(dummy)) {
                                found = dummy;
                                break;
                        }
                } while(cur != NULL);
        }


        FIFO_unlock(f);

        return found;
}

/*
 * FIFO_search2
 * Functioanlity: Search the FIFO "q" for an element that match the
 *              requirement specified in the callback subroutine "func", passing
 *              "arg" as the argument to "func"
 * Return value: an element if found; otherwise, NULL
 * Parameter:
 *              f - a pointer to FIFO
 *              func - a calback function "int func_name(fifo_queue_t *q,
 *                      void* ptr)" If the element matches, return 1; otherwise 0.
 */
fifo_queue_t * FIFO_search2(fifo_t *f,
                int (*func)(fifo_queue_t *, void *), void *arg) {
        fifo_queue_t *cur, *dummy, *found;

        if(f == NULL) return NULL;

        FIFO_lock(f);

        found = NULL;  /* Not found */
	if(f->count != 0) {
		cur = f->queue;

		do {
			dummy = cur;
			cur = cur->next;
			if(func(dummy, arg)) {
				found = dummy;
				break;
			}
		} while(cur != NULL);
	}


	FIFO_unlock(f);

	return found;
}

/*
 * FIFO_remove2
 * Functioanlity: Search the FIFO "q" for an element that match the
 *		requirement specified in the callback subroutine "func", passing
 *		"arg" as the argument to "func" and the remove the found element
 *		from the queue.
 * Return value: an element if found; otherwise, NULL
 * Parameter:
 *		f - a pointer to FIFO
 *		func - a calback function "int func_name(fifo_queue_t *q,
 *			void* ptr)" If the element matches, return 1; otherwise 0.
 */
fifo_queue_t * FIFO_remove2(fifo_t *f,
		int (*func)(fifo_queue_t *, void *), void *arg) {
	fifo_queue_t *prev, *cur, *dummy, *found;

	if(f == NULL) return NULL;

	FIFO_lock(f);

	found = NULL;  /* Not found */
	if(f->count != 0) {
		prev = cur = f->queue;

		do {
			if(func(cur, arg)) {
				found = cur;
				f->count--;

				if(prev == cur)
					f->queue = cur->next;
				else
					prev->next = cur->next;

				break;
			}
```

```
				if(prev != cur) prev = cur;
				cur = cur->next;
			} while(cur != NULL);
		}


		FIFO_unlock(f);

		return found;
}

/*
 * FIFO_foreach
 * Functioanlity: Walk through each element in FIFO and apply the
 *		callback routine "func" with each element.
 * Return value: NONE
 * Parameter:
 *		f - a pointer to FIFO
 *		func - a calback function "void func_name(fifo_queue_t *)"
 */
void FIFO_foreach(fifo_t *f, void (*func)(fifo_queue_t *)) {
		fifo_queue_t *cur, *dummy;

		if(f == NULL) return;

		FIFO_lock(f);

		if(f->count != 0) {
			cur = f->queue;

			do {
				dummy = cur;
				cur = cur->next;
				func(dummy);
			} while(cur != NULL);
		}

		FIFO_unlock(f);
}

/*
 * FIFO_foreach2
 * Functioanlity: Walk through each element in FIFO and apply the
 *		callback routine "func" with each element, passing arg as an
 *		additional parameter to "func"
 * Return value: NONE
 * Parameter:
 *		f - a pointer to FIFO
 *		func - a calback function "void func_name(fifo_queue_t *,
 *			void *arg)"
 *		arg - parameter to the callback function "func"
 */
void FIFO_foreach2(fifo_t *f,
		void (*func)(fifo_queue_t *, void *), void *arg) {
	fifo_queue_t *cur, *dummy;

	if(f == NULL) return;

	FIFO_lock(f);

	if(f->count != 0) {
		cur = f->queue;

		do {
			dummy = cur;
			cur = cur->next;
			func(dummy, arg);
		} while(cur != NULL);
	}

	FIFO_unlock(f);
}

/*
 * FIFO_lock
 * Functioanlity: Lock a FIFO
 * Return value: NONE
 * Parameter:
 *		f - a pointer to the FIFO to be locked
 */
void FIFO_lock(fifo_t *f) {
#ifndef __NO_FIFO_LOCK__
	int res;

	if(f == NULL) return;

#ifdef __PTHREAD__
	if( (res = pthread_mutex_lock(f->lock)) != 0 ) {
#else
	if( (res = _lwp_mutex_lock(f->lock)) != 0 ) {
#endif
		fprintf(stderr, "[FIFO_lock] _lwp_mutex_lock error\n");
		exit(-1);
	}
#endif
}

/*
 * FIFO_unlock
 * Functioanlity: Unlock a FIFO
 * Return value: NONE
 * Parameter:
 *		f - a pointer to the FIFO to be locked
 */
void FIFO_unlock(fifo_t *f) {
#ifndef __NO_FIFO_LOCK__
	int res;
```

```
        if(f == NULL) return;

#ifdef __PTHREAD__
        if( (res = pthread_mutex_unlock(f->lock)) != 0 ) {
#else
        if( (res = _lwp_mutex_unlock(f->lock)) != 0 ) {
#endif
                fprintf(stderr, "[FIFO_unlock] _lwp_mutex_unlock error\n");
                exit(-1);
        }
#endif
}
```

7.4) Server.c

```
/*
 * server.c : Skeleton multi-thread server
 * By: Athichart Tangpong
 *
 * Problem:
 *              1. The server can serve only 2037 connections. The server
 *                      can not create more than 2037 threads.
 *
 */

#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/types.h>
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "config.h"
#include "fifo.h"

#define  PORT 6001

typedef struct node_t node_t;
struct node_t {
        int node_id;
        char *hostname;
        fifo_t *keys;
};

// Global variable
extern char *optarg;
extern int errno;
fifo_t *node_queue;
```

```
// Prototype
void * thr_func(void *ptr);
int perform_sort(int fd);
void usage(char *st);
int CB_search_node_queue(fifo_queue_t *qq, void *node_id);
int CB_search_key(fifo_queue_t *qq, void *key);
int CB_search_key2(fifo_queue_t *qq, void *key);
int join(int fd, fifo_t *node_queue);
int leave(int fd, fifo_t *node_queue);
int search(int fd, fifo_t *node_queue);
int terminate(int fd, fifo_t *node_queue);

int main(int argc,char **argv) {
        struct  sockaddr_in cli_addr, serv_addr;
        int             sockfd, newsockfd;
        int             status, port, clilen;
        char            *delim="p:s:r:dv?";
        char            opt;
        fd_set  watchset;
        struct  timeval tv;

        port = PORT;
        while((opt = getopt(argc,argv,delim)) != EOF) {
                switch(opt) {
                        case 'p' : port = atoi(optarg); break;
                        case '?' :
                        default  : usage(argv[0]); exit(0);
                }
        }
        node_queue = FIFO_create();

        // Open a TCP socket (an Internet stream socket.)
        if( (sockfd=socket(AF_INET,SOCK_STREAM,0))<0 ) {
                perror("Server:can't open stream socket\n");
                exit(1);
        }

        // Bind our local address so that the client can send to us.
        bzero((char *) &serv_addr,sizeof(serv_addr));
        serv_addr.sin_family      = AF_INET;
        serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        serv_addr.sin_port             = htons(port);
        if( bind(sockfd,(struct sockaddr *) &serv_addr,
                                sizeof(serv_addr))<0) {
                perror("Server: can't bind local address\n");
                exit(1);
        }
        listen(sockfd,5);

        FD_ZERO(&watchset);
        FD_SET(sockfd, &watchset);
        tv.tv_sec = 10;
        tv.tv_usec = 0;

        for(;;) {
                pthread_t *th;
                int *para;
                int res;

                if(select(sockfd+1, &watchset, NULL, NULL, &tv) < 0) {
                        perror("Server: select error\n");
                        return -1;
                }

                if(FD_ISSET(sockfd, &watchset) != 0) {
                        clilen=sizeof(cli_addr);
                        if((newsockfd = accept(sockfd, (struct sockaddr *)
                                                &cli_addr,&clilen)) < 0) {
                                perror("Server: accept error\n");
                                return -1;
                        }

                        th = (pthread_t *)malloc(sizeof(pthread_t));
                        para = (int *)malloc(sizeof(int));
                        *para = newsockfd;

                        if( (res = pthread_create(th, NULL, thr_func,
                                                        (void *)para)) != 0) {
                                printf("Server: pthread_create error\n");
                                return -1;
                        }
                } else {
                        FD_SET(sockfd, &watchset);
                }
        }       /* for(;;) */
}

/*
 * thr_func
 * Functionality : Skeleton for the Multi-thread Server
 * Return Value : None
 * Parameter :
 *              ptr - pointer to the parameter
 */
void * thr_func(void *ptr) {
        int sockfd = *((int *)ptr);
        int flag;

        perform(sockfd);
        close(sockfd);
        free(ptr);

        return NULL;
}

// --------------------- Customized routines ----------------------

/*
 * CB_search_node_queue
 * Functionality : Search for the queue for the node "node_id"
 * Return Value : 1 if found, 0 otherwise.
 * Parameter :
 *              qq : an element in the look-up table which is a queue
 *                      of keys on a particular node
 *              node_id : the node_id to be searched for
 */
int CB_search_node_queue(fifo_queue_t *qq, void *node_id) {
        node_t *node;

        if(qq->ptr != NULL) {
                node = (node_t *)qq->ptr;
                if(node->node_id == *((int *)node_id))
                        return 1;
                else
                        return 0;
        }

        return 0;
}

/*
 * CB_search_key
 * Functionality : Search the look-up table for a node that
 *              hold a particular key
 * Return Value : 1 if found, 0 otherwise.
 * Parameter :
 *              qq : an element in the look-up table which is a queue
 *                      of keys on a particular node
 *              key : the key to be searched for
 */
int CB_search_key(fifo_queue_t *qq, void *key) {
        fifo_queue_t *qqq;
        node_t *node;

        if(qq->ptr != NULL) {
                node = (node_t *)qq->ptr;
                if( (qqq = FIFO_search2(node->keys, CB_search_key2, key))
                                != NULL ) {
                        return 1;
                } else {
                        return 0;
                }
        }

        return 0;
}

/*
 * CB_search_key2
 * Functionality : Search the queue for a node for a key
 * Return Value : 1 if found, 0 otherwise.
 * Parameter :
 *              qq : an element in the queue for a node which is a key
 *                key : the key to be searched for
 */
int CB_search_key2(fifo_queue_t *qq, void *key) {
        int l_key;

        if(qq->ptr != NULL) {
                l_key = *((int *)qq->ptr);
                if(l_key == *((int *)key)) {
                        return 1;
                } else {
                        return 0;
                }
        }

        return 0;
}

void CB_free_key(fifo_queue_t *qq) {
        int *key;

        if(qq->ptr != NULL) {
                key = (int *)qq->ptr;
                free(key);
        }
}

/*
 * perform
 * Functionality : Connection handler
 * Return Value : 1 if success, otherwise -1
 *      Parameter :
 *                fd - socket descriptor
 */
int perform(int fd) {
        short opcode;

        /* Receive opcode */
        if(readn(fd, &opcode, sizeof(short))<0) {
                perror("Server:readn error\n");
                return -1;
        }

        switch(opcode) {
                case JOIN : join(fd, node_queue); break;
                case SEARCH : search(fd, node_queue); break;
                case DEAD :
                case LEAVE: leave(fd, node_queue); break;
                case EXIT : terminate(fd, node_queue); break;
        }

        return 1;
}

/*
```

```
 * usage
 * Functionality : Print command-line usage
 * Return Value : None
 * Parameter :
 *       st - the program name
 */
void usage(char *st) {
        printf("Syntax : %s <option> <port>\n",st);
        printf("Valid option is\n");
        printf(" p <port>   : Server port\n");
        printf(" ?          : print this help message\n");
}

/*
 * join
 * Functionality : Handler for JOIN
 * Return Value : 1 if success, otherwise -1
 * Parameter :
 *              fd - socket descriptor
 *              node_queue - the look-up table
 */
int join(int fd, fifo_t *node_queue) {
        int  node_id, key_no, size, in_size;
        int *buffer, i, name_len;
        fifo_queue_t *qq;
        node_t *node;
        struct sockaddr_in name;

        name_len = sizeof(struct sockaddr);
        if(getpeername(fd, (struct sockaddr*)&name, &name_len)<0){
                 printf("Server: getpeername error\n");
                 return -1;
        }

        /* Receive node_id */
        if(readn(fd, &node_id, sizeof(int))<0) {
                perror("Server:readn error\n");
                return -1;
        }

        // check if the client already join
        if( (qq = FIFO_remove2(node_queue, CB_search_node_queue,
                                        (void *)&node_id)) != NULL ) {
                // Free the keys for the node
                printf("Node %d re-enter.\n", node_id);
                node = (node_t *)qq->ptr;
                FIFO_foreach(node->keys, CB_free_key);
                FIFO_destroy(node->keys);
                free(node->hostname);
                free(node);
                free(qq);
        }

        /* Receive number */
	if(readn(fd, &key_no, sizeof(int))<0) {
		perror("Server:readn error\n");
		return -1;
	}

	/* Receive key */
	buffer = (int *)malloc(key_no*sizeof(int));
	if(readn(fd, buffer, key_no*sizeof(int))<0) {
		perror("Server:readn error\n");
		return -1;
	}

	// Create a new queue for the joining node
	node = (node_t *)malloc(sizeof(node_t));
	node->node_id = node_id;
	node->hostname = (char *)strdup(inet_ntoa(name.sin_addr));
	node->keys = FIFO_create();
	printf("Node %d (%s) joined and published %d keys.\n",
			node_id, inet_ntoa(name.sin_addr), key_no);
	printf("keys: ");
	for(i=0; i<key_no; i++) {
		int *tmp = (int *)malloc(sizeof(int));

		*tmp = *(buffer)++;
		FIFO_enqueue(node->keys, FIFO_element(tmp));
		printf("%d ", *tmp);
	}
	printf("\n");
	FIFO_enqueue(node_queue, FIFO_element(node));

	printf("Numbers of Clients = %d\n", FIFO_count(node_queue));

	free(buffer);
	return 0;
}

/*
 * leave
 * Functionality : Handler for LEAVE
 * Return Value : 1 if success, otherwise -1
 * Parameter :
 *		fd - socket descriptor
 *		node_queue - the look-up table
 */
int leave(int fd, fifo_t *node_queue) {
	int	node_id;
	fifo_queue_t *qq;
	node_t *node;

	// Receive node_id
	if(readn(fd, &node_id, sizeof(int))<0) {
		perror("Server:readn error\n");
		return -1;
	}
```

```
        // Remove the node from node_queue
        if( (qq = FIFO_remove2(node_queue, CB_search_node_queue,
                                (void *)&node_id)) != NULL ) {
                // Free the keys for the node
                node = (node_t *)qq->ptr;
                FIFO_foreach(node->keys, CB_free_key);
                FIFO_destroy(node->keys);
                free(node->hostname);
                free(node);
                free(qq);
        }

        printf("Node %d left.\n", node_id);
        printf("Number of client = %d\n", FIFO_count(node_queue));

        return 0;
}

/*
 * search
 * Functionality : Handler for SEARCH
 * Return Value : 1 if success, otherwise -1
 * Parameter :
 *              fd - socket descriptor
 *              node_queue - the look-up table
 */
int search(int fd, fifo_t *node_queue) {
        int  node_id, key, result;
        fifo_queue_t *qq;
        node_t *node;

        // Receive node_id
        if(readn(fd, &node_id, sizeof(int))<0) {
                perror("Server:readn error\n");
                return -1;
        }

        // Receive number
        if(readn(fd, &key, sizeof(int))<0) {
                perror("Server:readn error\n");
                return -1;
        }

        // Search for the key
        result = -1;
        if( (qq = FIFO_search2(node_queue, CB_search_key,
                                (void *)&key)) != NULL ) {
                node = (node_t *)qq->ptr;
                result = node->node_id;
        }

        // Return search result
        if( writen(fd, &result, sizeof(int))<0 ) {
```

```
			printf("Server: writen error\n");
			return(-1);
		}


		if(result != -1) {
			int size;

			size = strlen(node->hostname);
			if( writen(fd, &size, sizeof(int)) < 0 ) {
				printf("Server: writen error\n");
				return(-1);
			}
			if( writen(fd, node->hostname, size) < 0 ) {
				printf("Server: writen error\n");
				return(-1);
			}

			printf("Node %d searched for the key %d and found on the client %d.\n",
					node_id, key, result);
		} else {
			printf("Node %d searched for the key %d and don't find\n\n ",
					node_id, key);
		}

		return 0;
}

/*
 * terminate
 * Functionality : Handler for EXIT (Terminate the server gracefully)
 * Return Value : 1 if success, otherwise -1
 * Parameter :
 *		fd - socket descriptor
 *		node_queue - the look-up table
 */
int terminate(int fd, fifo_t *node_queue) {
	// Free allocated resource
	exit(0);
}
```

7.5) tcp.c

```
/*
 * tcp.c : Basic TCP functions
 * By : Modified from Richard Steven's (Unix Network Programming)
 */

#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/lwp.h>
```

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

// Required by inet_addr()
#ifndef INADDR_NONE
#define INADDR_NONE 0xffffffff
#endif

// Required by gethostbyname_r()
#define BUFFER_SIZE             2048

lwp_mutex_t get_host_name;

/*
 * tcp_open (host,service,port)
 * Function  : Open TCP connection to the specified host at the
 *                 specified port
 * Return    : socket descriptor if OK, else -1 on error
 * Parameter :
 *        host - hostname or IP address
 *                 port - server's port
 */
int tcp_open (char *host,int port) {
        unsigned long inaddr;
        struct sockaddr_in tcp_srv_addr;                // Server's address
        struct hostent   *hp, *res;                             // For hostname lookup
        int fd, error;
        char buffer[BUFFER_SIZE];


        bzero ((char *)&tcp_srv_addr, sizeof(tcp_srv_addr));
        tcp_srv_addr.sin_family = AF_INET;
        tcp_srv_addr.sin_port = htons(port);

        hp = (struct hostent *)MALLOC(sizeof(struct hostent));

        // Determining host IP address or host name
        if((inaddr = inet_addr(host)) != INADDR_NONE) {
                // The address is in the dotted-decimal format
                bcopy ((char *)&inaddr, (char *)&tcp_srv_addr.sin_addr,
                        sizeof(inaddr));
        } else {
                // Lookup the hostname
//              _lwp_mutex_lock(get_host_name);

                if( (res = gethostbyname_r(host, hp, buffer,
                                2048, &error)) == NULL) {
                        printf("[tcp_open]: unable to find the host %s\n", host);
//                      _lwp_mutex_unlock(get_host_name);
                        return (-1);
                }

//              _lwp_mutex_unlock(get_host_name);
```

```
			// Hostname lookup succeeded
			bcopy (hp->h_addr, (char *)&tcp_srv_addr.sin_addr, hp->h_length);
		}

		// Create a socket
		if((fd=socket(AF_INET,SOCK_STREAM,0))<0) {
			printf("[tcp_open]: unable to create a socket\n");
			return(-1);
		}

		// Connect to the server
		if(connect(fd,(struct sockaddr *)&tcp_srv_addr,
					sizeof(tcp_srv_addr)) <0) {
			printf("[tcp_open]: unable to connect to server\n");
			close(fd);
			return(-1);
		}

		return(fd);
}

/*
 * readn (fd, ptr, nbytes)
 * Function  : read nbytes from file descriptor into ptr buffer.
 * Return    : number of read byte if success; otherwise, less than 0
 * Parameter :
 *		fd     = file (socket) descriptor
 *		ptr    = output buffer
 *		nbytes = number of bytes to be read
 */
int readn (int fd, char *ptr, long nbytes) {
  long nleft,nread;

  nleft=nbytes;
  while(nleft>0) {
    nread = read(fd,ptr,nleft);

    if(nread<0) return(nread);  /* error, return <0 */
    else if(nread==0) break;    /* EOF */

    nleft -= nread;
    ptr   += nread;
  }

  return(nbytes-nleft);         /* return >= 0 */
}

/*
 * writen (fd, ptr, nbytes)
 * Function  : write nbytes from ptr buffer to file descriptor.
 * Return    : number of written byte if success; otherwise, less than 0
 * Parameter :
 *		fd     = file (socket) descriptor
 *                    ptr    = input buffer
 *                    nbytes = number of bytes to be written
 */
int writen (int fd, char *ptr, long nbytes) {
  long nleft,nwritten;

  nleft = nbytes;
  while(nleft>0) {
    nwritten = write(fd,ptr,nleft);

    if(nwritten<=0) return(nwritten);

    nleft -= nwritten;
    ptr   += nwritten;
  }

  return(nbytes-nleft);
}

/*
 * readline (fd, ptr, maxlen)
 * Function  : read 1 line or maxlen bytes from fd into prt
 * Return    : number of read byte if success; otherwise, less than 0
 * Parameter :
 *                    fd     = file (socket) descriptor
 *                    ptr    = output buffer
 *                    maxlen = the maximum number of bytes to be written
 */
int readline (int fd, char *ptr, int maxlen) {
  int n,rc;
  char c;

  for(n=1;n<maxlen;n++) {
    if((rc=read(fd,&c,1))==1) {
       *ptr++ = c;
      if(c=='\n') break;
    }
    else
                      if(rc==0) {
                                if(n==1) return(0);
                                else break;
                      } else return(-1);
  }

   *ptr = 0;
  return(n);
}
```

7.6) scanner.c

```
#include <stdio.h>
#include "config.h"
#include "fifo.h"
#include "scanner.h"
```

```
// Global variables
char lexeme[MAX_TOKEN_LEN + 1];
FILE *yyin;

// Prototypes
char *expect(int expected);

char *title()           { return (char *)expect(RTITLE); }
char *artist()    { return (char *)expect(RARTIST); }
char *country() { return (char *)expect(RCOUNTRY); }
char *company()         { return (char *)expect(RCOMPANY); }
char *year()            { return (char *)expect(RYEAR); }

char *expect(int expected) {
	int token, first = 1;
	char *ptr, *buffer = (char *)malloc(MAX_TOKEN_LEN + 1);

	sprintf(buffer, "");
	while((token = yylex()) != expected) {
		if(!first) {
			ptr = (char *)strdup(" ");
			strcat(buffer, ptr);
		} else {
			first = 0;
		}

		ptr = (char *)strdup(lexeme);
		strcat(buffer, ptr);
	}

	return buffer;
}

int read_data(char *filename, fifo_t *key_list) {
	int token;
	keys_t *object;

	yyin = fopen(filename, "r");

	token = yylex();
	while(token != 0) {
		if((token == TEXT) && !strcmp(lexeme, "key")){
			object = (keys_t *)malloc(sizeof(key_t));

			if((token = yylex()) == NUMBER) {
				object->key = atoi(lexeme);
				printf("Key = %d\n", object->key);
			}

			token = yylex();
			while((token != 0) && strcmp(lexeme, "key")) {
				switch(token ) {
					case LTITLE : object->title = title(); break;
					case LARTIST : object->artist = artist(); break;
```

```
                                        case LCOUNTRY : object->country = country();
break;
                                        case LCOMPANY : object->company = company();
break;
                                        case LYEAR : object->year = year(); break;
                                }

                                token = yylex();
                        }

                        FIFO_enqueue(key_list, FIFO_element(object));

                        printf("Title = %s\n", object->title);
                        printf("ARTIST = %s\n", object->artist);
                        printf("COUNTRY = %s\n", object->country);
                        printf("COMPANY = %s\n", object->company);
                        printf("YEAR = %s\n", object->year);
                        printf("\n");
                }
        }

        return 0;
}

/*
 * test_parser
 * Functionality : Test the parser. No use in any program
 * Return Value : NONE
 * Parameter :
 *      filename - the file to be parsed
 */
void test_parser(char *filename) {
        int token;

        yyin = fopen(filename, "r");

        while((token=yylex()) != 0) {
                switch(token) {
                        case TEXT : printf("TEXT\t\t"); break;
                        case NUMBER : printf("NUMBER\t\t"); break;
                        case LTITLE : printf("LTITLE\t\t"); break;
                        case RTITLE : printf("RTITLE\t\t"); break;
                        case LARTIST : printf("LARTIST\t\t"); break;
                        case RARTIST : printf("RARTIST\t\t"); break;
                        case LCOUNTRY : printf("LCOUNTRY\t\t"); break;
                        case RCOUNTRY : printf("RCOUNTRY\t\t"); break;
                        case LCOMPANY : printf("LCOMPANY\t\t"); break;
                        case RCOMPANY : printf("RCOMPANY\t\t"); break;
                        case LYEAR : printf("LYEAR\t\t"); break;
                        case RYEAR : printf("RYEAR\t\t"); break;
                        case ' ' : continue;
                }
                printf("%s\n", lexeme);
        }
}
```

7.7) scanner.l

```
%{
#include <string.h>
#include "scanner.h"

extern char lexeme[];
%}

%%
[ \t\n]         ;
[ ]                     {
                                        strcpy(lexeme, yytext);
                                        return yytext[0];
                                }
[a-zA-Z]*       {
                                        strcpy(lexeme, yytext);
                                        return TEXT;
                                }
[0-9]*          {
                                        strcpy(lexeme, yytext);
                                        return NUMBER;
                                }
[<]TITLE[>]     {
                                        strcpy(lexeme, yytext);
                                        return LTITLE;
                                }
[<][/]TITLE[>] {
                                        strcpy(lexeme, yytext);
                                        return RTITLE;
                                }
[<]ARTIST[>]  {
                                        strcpy(lexeme, yytext);
                                        return LARTIST;
                                }
[<][/]ARTIST[>]         {
                                        strcpy(lexeme, yytext);
                                        return RARTIST;
                                }
[<]COUNTRY[>]           {
                                        strcpy(lexeme, yytext);
                                        return LCOUNTRY;
                                }
[<][/]COUNTRY[>]        {
                                        strcpy(lexeme, yytext);
                                        return RCOUNTRY;
                                }
[<]COMPANY[>]           {
                                        strcpy(lexeme, yytext);
                                        return LCOMPANY;
                                }
[<][/]COMPANY[>]        {
                                        strcpy(lexeme, yytext);
                                        return RCOMPANY;
                                }
```

```
[<]YEAR[>]		{
					strcpy(lexeme, yytext);
					return LYEAR;
				}
[<][/]YEAR[>]		{
					strcpy(lexeme, yytext);
					return RYEAR;
				}

%%
yywrap() {
	return 1;
}
```

7.8) config.h

```
#ifndef __CONFIG_H__
#define __CONFIG_H__

// Opcodes of the protocol
#define JOIN	1
#define SEARCH		2
#define LEAVE	3
#define SREPLY	4
#define GET		5
#define GREPLY		6
#define DEAD	7
#define EXIT	20

typedef struct keys_t keys_t;
struct keys_t {
	int key;
	char *title;
	char *artist;
	char *country;
	char *company;
	char *year;
};

#endif
```

7.9) fifo.h

```
/*
 * fifo.h
 * FIFO with mutex lock
 *	$Id: fifo.h,v 1.1 2002/11/18 06:07:49 tangpong Exp $
 */

#ifndef __FIFO_H__
#define __FIFO_H__

#include <sys/lwp.h>
```

```
/* An ordinary singly link list */
typedef struct queue_s fifo_queue_t;
struct queue_s {
	void *ptr;			/* Content */
	fifo_queue_t * next;		/* Next element */
};

/* FIFO */
typedef struct {
	unsigned int count;			/* Number of elements in the FIFO */
	void *ext;				/* Customized datas structure */
#ifdef __PTHREAD__
	pthread_mutex_t *lock;
#else
	lwp_mutex_t *lock;			/* Mutex lock for the FIFO */
#endif
	fifo_queue_t *queue;			/* Queue */
} fifo_t;

/* Utilities */
void * MALLOC(int size);
void FREE(void *ptr);

fifo_t * FIFO_create();
void FIFO_destroy(fifo_t *);
fifo_queue_t * FIFO_element(void *ptr);		// ptr must be allocated
int FIFO_is_empty(fifo_t *);
int FIFO_count(fifo_t *f);
void FIFO_enqueue(fifo_t *, fifo_queue_t *);
fifo_queue_t * FIFO_dequeue(fifo_t *);
fifo_queue_t * FIFO_search(fifo_t *, int (*func)(fifo_queue_t *));
fifo_queue_t * FIFO_search2(fifo_t *, int (*func)(fifo_queue_t *, void *), void *arg);
fifo_queue_t * FIFO_remove2(fifo_t *, int (*func)(fifo_queue_t *, void *), void *arg);
void FIFO_foreach(fifo_t *, void (*func)(fifo_queue_t *));
void FIFO_foreach2(fifo_t *f, void (*func)(fifo_queue_t *, void *), void *arg);
void FIFO_lock(fifo_t *);
void FIFO_unlock(fifo_t *);

#endif
```

7.10) tcp.h

```
/*
 * tcp.h
 */

#ifdef __TCP_H__

int tcp_open(char *host, int port);
int readn(int fd, char *ptr, long nbytes);
int writen (int fd, char *ptr, long nbytes);
int readline (int fd, char *ptr, int maxlen);

#endif
```

7.11) scanner.h

```
#define TEXT            257
#define NUMBER          258
#define LTITLE          259
#define RTITLE          260
#define LARTIST         261
#define RARTIST         262
#define LCOUNTRY        263
#define RCOUNTRY        264
#define LCOMPANY        265
#define RCOMPANY        266
#define LYEAR           267
#define RYEAR           268

#define MAX_TOKEN_LEN         256
```

7.12) Makefile

```
CC              = gcc
CFLAGS          = -g -D_REENTRANT -D__PTHREAD__
CFLAGS          += $(INCLUDES)
LEX             = lex
LIBS            += -lnsl -lsocket -lpthread
INCLUDES        = -I./include
SRC             = ./src
SERVER          = $(SRC)/server.c $(SRC)/tcp.c $(SRC)/fifo.c
CLIENT  = $(SRC)/client.c $(SRC)/tcp.c $(SRC)/fifo.c $(SRC)/lex.yy.c $(SRC)/scanner.c

main: server client

server:
	$(CC) $(CFLAGS) $(LIBS) -o svc $(SERVER)

client:
	$(LEX) $(SRC)/scanner.l
	mv lex.yy.c $(SRC)
	$(CC) $(CFLAGS) $(LIBS) -o cli $(CLIENT)

clean:
	rm -f a.out *.o cli svc
```