# DISTRIBUTED LICENSE SERVER

By

SOPANA PROHMBHADRA

Bachelor of Science

Thammasat University

Bangkok, Thailand

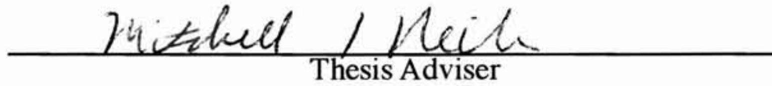
1991

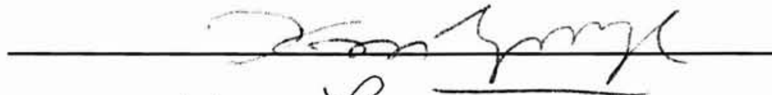
Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1995

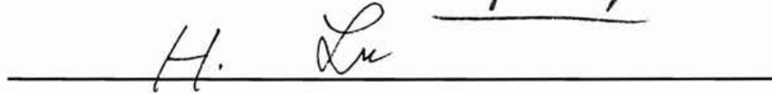
OKLAHOMA STATE UNIVERSITY

**DISTRIBUTED LICENSE SERVER**

Thesis Approved:


Thesis Adviser





Dean of the Graduate College

**ACKNOWLEDGMENTS**

I profoundly thank my major advisor, Dr. Mitch Neilsen for his constant help, guidance, warm encouragement, inspiration and friendship. He patiently guided me throughout my thesis, always helping me when needed and giving me exemplary advice, without which I couldn't have completed my thesis successfully. My sincere appreciation extends to my other committee members Dr. K. M. George and Dr. H. Lu, whose guidance, assistance, and encouragement are also invaluable.

Moreover, I wish to express my sincere thanks to those who provided suggestions and assistance for this study: Dr. J. P. Chandler, Dr. M. H. Samadzadeh, Dr. J. La France, and Dr. B. Mayfield. My special thanks to Rikip Ginanjar who helped and gave me the valuable advice.

Throughout my life, my parents and my two sisters provide constant support, love, understanding, and strong encouragement all the time so I would also like to give my special and deepest appreciation to them.

Finally, I would like to thank the Department of Computer Science for supporting me during these two years of study.

## TABLE OF CONTENTS


| Chapter | Page |
|---|---|
| I. INTRODUCTION | 1 |
| II. BASIC CONCEPTS AND DEFINITIONS | 3 |
| Coterie | 3 |
| K-Coteries | 3 |
| Nondominated K-Coteries | 4 |
| III. VOTE ASSIGNMENT ALGORITHM | 5 |
| IV. DESIGN AND IMPLEMENTATION | 7 |
| System Initialization | 8 |
| Run-time Algorithm | 9 |
| Lamport's Logical Clocks | 9 |
| Distributed K-Mutual Exclusion Algorithm | 11 |
| V. MODIFICATIONS TO ENHANCE PERFORMANCE | 13 |
| VI. PERFORMANCE ANALYSIS | 15 |
| VII. CONCLUSIONS AND FUTURE WORK | 17 |
| Conclusions | 17 |
| Future Work | 17 |
| BIBLIOGRAPHY | 18 |
| APPENDIXES | 19 |
| APPENDIX A--HEADER FILE | 20 |
| APPENDIX B--PROGRAM GENERATED K-COTERIE | 24 |
| APPENDIX C--DISTRIBUTED LICENSE SERVER | 43 |

## LIST OF FIGURES


| Figure | Page |
|---|---|
| 1. Distributed license server | 7 |
| 2. System initialization | 8 |
| 3. Lamport's logical clocks | 10 |

# CHAPTER I

# INTRODUCTION

Generally, in existing distributed systems, a centralized license server is used to service particular requests of users to use a floating license for a software package. When a user wishes to use a license, the user sends a request to a centralized license server which controls access to the licenses. If the server crashes, no users can obtain a license. This single point of failure is a serious drawback. A distributed license server can overcome this disadvantage. It can be used to control access to floating licenses, and it can increase the reliability of the system, as well.

In a distributed license server, there are N servers that contain identical license resources shared by nodes or users, and k licenses that can be used by users simultaneously.

The purpose of this thesis is to design and implement a distributed license server by using the concept of k-coteries and a distributed k-mutual exclusion algorithm to control access to the licenses. The server allows at most k processes to obtain a license and use the software package at the same time. In order to construct the nondominated k-coterie used by the servers, the vote assignment algorithm proposed by Ginanjar [G95] is used. Furthermore, in order to control access to the licenses, the distributed k-mutual exclusion algorithm of Kakugawa, Fujita, Yamashita, and Ae [KFYA95] is used in this thesis. This algorithm is a permission-based algorithm in which a process wishing to enter a critical section (obtain a license) asks other processes for permission and retains permission until it leaves the critical section.

This thesis contains seven chapters. The second chapter briefly reviews basic concepts and definitions including quorum, coterie, and k-coterie. The vote assignment algorithm of Ginanjar is presented in Chapter III. The design implementation of a distributed license server is discussed in Chapter IV. The concept of Lamport's logical clocks is also presented. Then, modifications to enhance performance is discussed in Chapter V. The performance of the system is analyzed in Chapter VI. Finally, conclusions and future work are included in Chapter VII.

# CHAPTER II

# BASIC CONCEPTS AND DEFINITIONS

The followings are the concepts and definitions of some terms used in this thesis [FY91, KFYA93, NM94]. Let U denote a non-empty set of nodes in a distributed system.
A **quorum** is a collection of set of nodes in the system.
A **coterie** is a collection of quorums; in particular, Q is a coterie under U if it satisfies the following conditions:

(1) If $G \in Q$, then $G \neq \varnothing$ and $G \subseteq U$;

(2) Minimality Property: for any two distinct quorums $G, H \subseteq Q$, $G \not\subset H$;

(3) Intersection Property: if $G, H \in Q$, then $G \cap H \neq \varnothing$.

A **k-coterie** is an extension of a coterie. A k-coterie is a collection of k pairwise disjoint quorums. Formally, Q is a k-coterie under U if the following three conditions hold:

(1) Minimality Property:

For any two distinct quorums $G, H \in Q$, $G \not\subset H$.

(2) Intersection Property:

For any (k+1) quorums $G_1,\ldots,G_{k+1} \in Q$, there exists a pair of quorums intersecting each other.

(3) Non-intersection Property:

For any $(h < k)$ and any set of h pairwise disjoint quorums $D = \{G_1, \ldots G_h\} \subseteq Q$; there exists a quorum $G \in Q$ such that $G \cap G_i = \varnothing$ for all $G_i \in D$.

Property (2) guarantees that at most k processes can enter their critical sections (e.g., obtain a license) at the same time. Moreover, by property (3), if less than k processes have obtained licenses, then a process wishing to obtain a license can do so by selecting an appropriate quorum.

**Dominated and Nondominated k-coteries**

Let Q1 and Q2 be k-coteries under U. Then Q1 **dominates** Q2 if

(1) $Q_1 \neq Q_2$

(2) $(\forall H \in Q_2)$ $[\exists G \in Q_1$ such that $G \subseteq H]$

If there is no such k-coterie, then Q is a **nondominated k-coterie.**

# CHAPTER III

# VOTE ASSIGNMENT ALGORITHM

Neilsen and Mizuno have shown that nondominated k-coteries are the most resilient to network and site failures [NM94], so we focus on nondominated k-coteries. Since we assume that there are N identical license servers with at most k licenses, we need an algorithm that can form a nondominated k-coterie for any values of N and k.

Ginanjar proposed a vote assignment algorithm that can be used to construct a nondominated k-coterie for any values of N and k [G95].

## Vote Assignment Algorithm

Let S be the set of N nodes (license servers) in the system, and let k be an integer ($1 \le k \le N$). Let $v$ denotes a vote assignment function from S to Z ($v$: S $\rightarrow$ Z, where Z is the set of nonnegative integers).

For a vote assignment $v$ over S, MAJ (majority) and TOT (total) are defined by

$$MAJ = \lceil (N+1)/(k+1) \rceil$$

$$TOT = (k+1)MAJ - 1$$

If N = TOT, every node has the same number of votes: one vote; otherwise, let $M$ be a subset of S such that $|M|$ = TOT - N. Every node in $M$ has two votes, the others have one vote.

Let $v(a)$ be the number of votes assigned to node $a$; $a \in$ S.

$$v(a) = \begin{cases} 2 & \forall a \in M \\ 1 & \forall a \in S - M \end{cases}$$

Nodes with more than one vote have more power than the others.

A subset G in S is called a **quorum** if the number of votes assigned to G is:

$$v(G) = \begin{cases} MAJ + 1 & \text{if } MAJ \text{ is odd and } G \subseteq M \\ MAJ & \text{otherwise} \end{cases}$$

A **k-coterie** Q is the collection of all quorums.

For example, let S = {1,2,3,4,5}. That is, N = 5. Let k = 3. Then, MAJ = 2, and TOT = 7. Since TOT > N, $|M|$ = TOT - N = 2. Assume that, {1,2} $\in$ M, then nodes in {1,2} have two votes, and nodes in {3,4,5} have one vote. The resulting k-coterie is:

{{1}, {2}, {3,4}, {3,5}, {4,5}}

A program to create k-coteries using this approach is shown in Appendix B.

# CHAPTER IV

## DESIGN AND IMPLEMENTATION

In this implementation, we assume that there are N identical license servers with at most k licenses; N and k are integers, and $N \geq k$. These license servers are shared among nodes (users) in a distributed system. Each node can access at most one license at a time, and each license server can be accessed by at most one node at a time. Since there are k licenses, at most k nodes can access k license servers simultaneously. In order to control access to the licenses, we use the concept of k-coteries, which is generally used to solve k-mutual exclusion problems, and distributed k-mutual exclusion algorithm.

Assume that, on each license server, there are 2 interfaces: a client interface (called the local license manager or LLM) and a server interface (called the remote license controller or RLC). An LLM controls access to RLCs while an RLC services requests sent from other LLMs as shown in Figure 1.

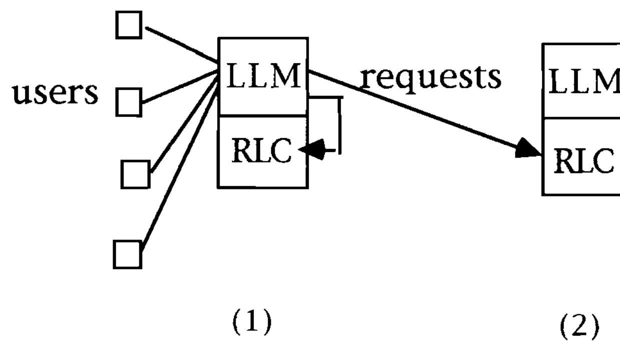



Figure 1. Distributed license servers.

**System Initialization**

When a license server is initialized, the LLM sends messages to all RLCs on all license servers and checks if any of them are running (the addresses of all license servers and the maximum number of licenses (k) will be kept in a file which will be copied and kept on every server.) Before a user can successfully connect to a license server, all license servers must be started up. That is, all LLMs must be able to communicate with all RLCs in the system as shown in Figure 2. Each LLM then forms a k-coterie by using the vote assignment algorithm proposed by Ginanjar [G95]. This k-coterie is identical to the k-coteries formed by all other LLMs. The LLM uses the k-coterie to determine which quorum of servers should be contacted to obtain a license. Each RLC is allowed to grant permission to one LLM at a time.

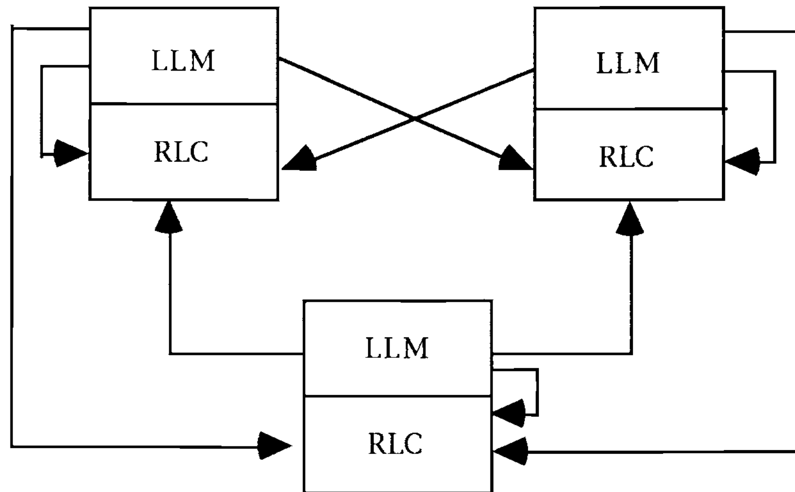



Figure 2. System initialization.

Each LLM has local variables YES and NOTNOW. YES is a queue keeping RLC processes which have sent permissions (by message OK) to the LLM. NOTNOW keeps RLCs sending WAIT messages to indicate that they could not give permissions to the LLM.

An RLC also has its local variables: PERM and QUEUE. PERM keeps the LLM which has received an OK from this RLC, but has not sent a RELEASE message to tell that a user has finished using the software. Since an RLC never gives permission to two LLMs at one time, PERM can be empty or contains only one LLM. QUEUE keeps requests from LLMs in timestamp order.

Initially, these four variables: YES, NOTNOW, PERM, and QUEUE are set to be nil or empty.

**Run-time Algorithms**

Let $R_i$ be the RLC at server i, and $L_i$ be the LLM at server i. When a user wants to obtain a license, a request is sent to $L_i$. The license server $L_i$ keeps this request on a queue (called the user queue). In order to avoid deadlocks and starvation, a timestamp (t, p) is attached to each request. Time t is the logical time at which a user initiates a request, and p is a system-wide unique identifier for the LLM [KFYA95]. Since p is unique, so is the timestamp (t, p). In order to calculate logical time t, we use **Lamport's logical clocks** [SS94] as follow:

Let $C_i$ be a clock at each process $P_i$ ; *a*, *b* are events; $C_i$ (*a*) is a function related to any event *a*. The logical time is:

(1) if *a* and *b* are two successive events in $P_i$ , and *a* occurs before *b* denoted

$(a \rightarrow b)$, then

$C_i(b) = C_i(a) + d \qquad (d > 0);$

That is, if $a \rightarrow b$, then $C_i(a) < C_i(b)$. Usually, d is equal to 1.

(2) Let *a* be the event of sending a message in $P_i$, and *b* be the corresponding event of receiving the message in $P_k$, then

$$C_i(a) < C_k(b).$$

When $P_k$ receives a message, $C_k$ is set to a value greater or equal to its present value and greater than $C_i$.

$$C_k(b) = \max(C_k(b), C_i(a) + d) \qquad (d > 0)$$

For example:

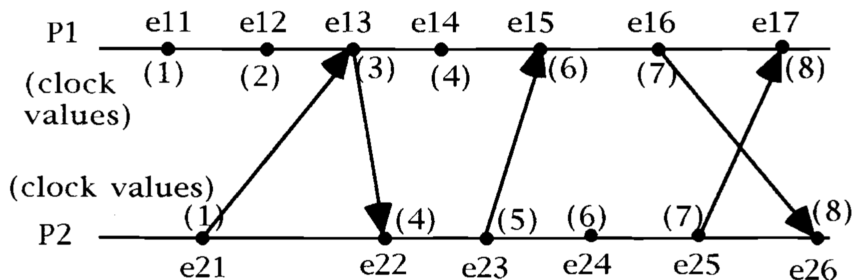



figure 3. Lamport's logical clocks.

From Figure 3, let d = 1, and $C_i(k)$ denotes the clock related to event k at $P_i$. Initially, $C_1$ and $C_2$ are set to zero. When $e_{11}$ occurs, $C_1$ is incremented by 1 ($C_1(e_{11}) = 1$). Since $e_{11}$ and $e_{12}$ are two successive events, $C_1(e_{12}) = 2$. Since $P_2$ sends a message from $e_{21}$ to $e_{13}$ of $P_1$, $C_1(e_{13}) = \max(C_1(e_{13}), C_2(e_{21}) + 1) = \max(3, 1+1) = 3$, and the logical time at $e_{22}$ of $P_2$ is $C_2(e_{22}) = \max(2, 3+1) = 4$, and so on.

After receiving a request, $L_i$ will send requests to RLCs using the quorums passed in the initialization step. Since the distributed k-mutual exclusion algorithm (k-mutex algorithm) proposed by Kakugawa, Fujita, Yamashita, and Ae [KFYA93, KFYA95] is a permission-based algorithm which uses the concept of k-coteries to allow at most k

processes to obtain at most k licenses at the same time, we applied and modified this k-mutex algorithm in this thesis. The modified algorithm is:

Let X be a k-coterie under the set $S = \{R_1, R_2, ..., R_N\}$, and Q be a quorum in X.

- When receiving a request from a user, $L_i$ submits a request (t, $L_i$) to each member (RLCs) of a quorum Q in X and waits for an OK or WAIT message from those RLCs.

- If $L_i$ has gathered permission from each member in Q, $L_i$ sends permission to the user. After that, the user uses the licensed software. Upon termination, a FINISH message is sent to $L_i$. Then, $L_i$ sends a RELEASE to all members in Q.

- If some RLCs in Q answer WAIT, $L_i$ adds the RLCs answering OK to YES and adds the RLCs sending WAIT messages to NOTNOW. After that, $L_i$ selects another quorum Q´ in X whose members must not be in NOTNOW, and Q´ must be a quorum maximizing $|Q' \cap YES|$; repeat the procedure from the first step. If there is no such Q´ quorum satisfying the condition, $L_i$ has to wait for OK messages from RLCs in NOTNOW.

- If an RLC in NOTNOW sends an OK to $L_i$ during the above procedure, $L_i$ moves this RLC from NOTNOW to YES and check if there is a quorum whose members are included in YES. If the test is successful, $L_i$ sends a message with an address of a license server (randomly chosen) in the selected quorum to a user in the user queue. The user then can use the software.

- When user finishes using the software, he sends a FINISH message to $L_i$. Next, $L_i$ sends a RELEASE to each RLC in $YES \cap Q$. If at that time, there is no more request from users, $L_i$ sends RELEASE messages to all RLCs in $YES \cup NOTNOW$.

- When RLC gets a request (t, $L_i$) from $L_i$, if PERM is empty, RLC sends an OK to $L_i$ and inserts the request to PERM.

- If PERM (with request ($t_p$, $L_p$)) is not empty, RLC adds the request (t, $L_i$) in QUEUE. After that, RLC checks whether the request (t, $L_i$) is the smallest timestamp (the highest priority) among those in QUEUE and PERM. Let request ($t_Q$, $L_Q$) be the smallest

timestamp in QUEUE. If $(t, L_i) > \min \{(t_Q, L_Q), (t_p, L_p)\}$ , RLC sends a WAIT to $L_i$. Otherwise, RLC sends QUERY message to $L_p$ in PERM in order to ask if $L_p$ has already got a license; wait for an answer (RELINQUISH or RELEASE message) from $L_p$.

- If RLC gets a RELINQUISH, it moves request $(t_p, L_p)$ from PERM to QUEUE in the order of timestamp and moves request $(t, L_i)$ from QUEUE to PERM. Finally, RLC sends a WAIT to $L_p$ and sends an OK to $L_i$.

- When RLC gets a RELEASE from $L_i$, it removes the request $(t, L_i)$ from PERM. If QUEUE is not empty, RLC moves the smallest timestamp $(t_Q, L_Q)$ (the highest priority) from QUEUE to PERM and sends an OK to $L_Q$.

- When $L_i$ gets a QUERY message from RLC, it checks if it has successfully locked all RLCs in a selected quorum Q (getting a license). If not, $L_i$ moves this RLC from YES to NOTNOW and sends a RELINQUISH back to RLC. $L_i$ does nothing if $L_i$ has got a license, or RLC is not in YES.

The program code to implement the distributed license server is included in Appendix C.

# CHAPTER V

# MODIFICATIONS TO ENHANCE PERFORMANCE

The algorithm mentioned in the previous chapter is not exactly the same as the k-mutex algorithm. When an LLM cannot obtain a license after sending requests to all RLCs in Q, it selects a new quorum Q´ which satisfies the condition: Q´ must be a quorum maximizing $|Q' \cap YES|$, and its members must not be in NOTNOW. In the k-mutex algorithm, LLM sends requests only to member in (Q´ - YES). On the other hand, in our algorithm, LLM sends requests to all RLCs in Q´.

- If there are more than one user connecting to an LLM, and these users asks for a license at the same time, the LLM keeps all requests from these users on the user queue. After that, the LLM sends requests to all RLCs in Q or Q´. After getting a license, the LLM may try to collect permission from RLCs in another quorum in order to get another license for another user. Then, it is possible for the LLM to send two requests to an RLC: one may be in PERM, the other may be in QUEUE. These two requests are unique since each request is a unique timestamp (t, p). Note that while the RLC keeps the request in QUEUE, the LLM keeps this RLC in NOTNOW, then there can be at most one request from one LLM keeping in QUEUE of RLCs since the selected Q´ must not be in NOTNOW.

For example, assume that there are 5 license servers with 2 licenses. That is, N = 5, and k = 2. The k-coterie can be

{{1,2}, {1,3}, {1,4}, {1,5}, {2,3}, {2,4}, {2,5}, {3,4}, {3,5}, {4,5}}

Let $LLM_i$ be the local license manager i, and $RLC_i$ be the remote license controller i. If there are 3 users ($U_1$, $U_2$, $U_3$) connecting to $LLM_1$. All of them request to use a license. $LLM_1$ keeps all requests from these users on the user queue. If $LLM_1$ can obtain a license after gathering permission from all RLCs in {1,2}, it keeps $RLC_1$ and $RLC_2$ on YES while $RLC_1$ and $RLC_2$ keep $LLM_1$ in PERM. Since there are more than one request

on the user queue, $LLM_1$ tries to collect permission from another quorum. Assume that quorum {1,3} is chosen. $LLM_1$ sends requests to both $RLC_1$ and $RLC_3$. $RLC_3$ may send back OK or WAIT message to $LLM_1$ while $RLC_1$ keeps the request in QUEUE and sends WAIT message back to $LLM_1$ because PERM is not empty.

We can conclude that in the modified algorithm, multiple users connected to a license server can use the license software simultaneously if there is a license available without waiting for another user to finish its job.

- If LLM has already obtained a license, but a user sent a request to LLM has already left the system, LLM can pass this license to another user with the highest priority. If there are no more requests in the user queue, the LLM sends RELEASE messages to all RLCs in YES and NOTNOW.

In this case, when a user leaves the system, we do not have to send RELEASE messages to all RLCs in Q or Q′, select a new quorum, and gather permission from RLCs in this new quorum if there is another request in the user queue. This aspect can enhance the performance of the system.

# CHAPTER VI

# PERFORMANCE ANALYSIS

In this thesis, we use the concept of k-coterie to implement a distributed license server, then we focused on k-coterie qualification. Since the (average) number of messages necessary to get a license seems to be in proportion to the size of quorums [KYA91], we can use the size of quorum to measure the best case performance of a distributed license server.

The vote assignment algorithm of Ginanjar [G95] mentioned above creates a k-coterie which contains k pairwise disjoint quorums. Let N be the total number of license servers in the system. Each node in a system has one vote if N = TOT. Otherwise, $|M|$ = TOT - N; M is a subset of S, and every node in M has two votes, the others have one vote. The total votes in each quorum G ($v$(G)) are equal MAJ + 1 if MAJ is odd, and G $\subseteq$ M; otherwise, $v$(G) is equal MAJ. Then, size of a quorum containing exactly one element of M is ($v$(G) - 1). If a quorum contains two elements of M, the size of this quorum is ($v$(G) - 2), and so on. Then, quorums contain h elements of M have the size ($v$(G) - h). Since $MAJ = \lceil (N+1)/(k+1) \rceil$, then the size of each quorum of k-coterie produced is O(N/k).

From the k-mutex algorithm of Kakugawa, Fujita, Yamashita, and Ae [KFYA95], when a user sends a request to a license server, quorum Q is selected. LLM sends REQUESTs to all RLCs in Q. If all RLCs return OK messages to the LLM, LLM can get a license. After the user finishes using the software, LLM sends RELEASE messages to all RLCs. In this best case, the total number of messages transmitted between an LLM and RLCs are 3($v$(G) - h)), where ($v$(G) - h) is the size of Q. In the worst case, 6N messages are required to transmitted to get a license. For example, when $LLM_i$ sends a request to $RLC_j$ in Q, $RLC_j$ may send QUERY to $LLM_k$, and $LLM_k$ may send RELINQUISH to $RLC_j$. Next, $RLC_j$ sends OK to $LLM_i$. $LLM_i$ then sends RELEASE to $RLC_j$. Finally, $RLC_j$ sends OK to $LLM_k$.

In the case that LLM could not get a license after sending REQUESTs to all RLCs in Q, the maximum number of messages transmitted is $2(\nu(G) - h)$; the total consists of $(\nu(G) - h)$ request messages from LLM and $(\nu(G) - h)$ WAIT or OK messages from RLCs in Q. After that, LLM selects another quorum Q′ which satisfies the following conditions: Q′ minimizes $|Q' \cap YES|$ which does not intersect with NOTNOW. If LLM still fails to collect permission from all RLCs in Q′, it continues selecting a quorum Q′ until it cannot find Q′ satisfying the above condition. Then, LLM fails to get a license.

# CHAPTER VII

# CONCLUSIONS AND FUTURE WORK

## Conclusions

In order to overcome the single point of failure of the centralized license server, the distributed license server has been implemented by using the concept of k-coteries and the distributed k-mutual exclusion algorithm. The vote assignment algorithm of Ginanjar [G95] is used to create the nondominated k-coterie which allows at most k processes to obtain k licenses at the same time. The k-mutex algorithm of Kakugawa, Fujita, Yamashita, and Ae [KFYA95] is applied and modified to work with this system. Moreover, the logical clock concept of Lamport [SS94] is used to calculate the logical time. Modifications to enhance performance is discussed, and the performance of this system is also analyzed. In order to obtain a license, the total number of messages necessary is $3(\nu(G) - h)$ in the best case, but in the worst case, the total number of messages transmitted is $O(N)$.

## Future Works

In this implementation, at the system initialization step, we assume that all license servers must be started up. However, if some license servers are not running, we may check how many license servers are running at that time and form a k-coterie from these running servers. If there is a license server starting after forming a k-coterie, we may form a new k-coterie from the new total number of running license servers. Before adjusting the k-coterie, we may have to check if there is any request from users. If not, we can adjust the k-coterie. Otherwise, we may have to wait until there is no request from users or LLM has not used the previous k-coterie.

# BIBLIOGRAPHY

[FY91] S. Fujita, and M. Yamashita. Constructing asymptotically optimal k-coteries. In Proceedings of the 2nd International Symposium on Algorithms, pages 1-25, 1991.

[G95] R. Ginanjar. A method for constructing nondominated k-coteries. Oklahoma State University Masters Thesis, 1995.

[KYA91] S. Fujita, M. Yamashita, and T. Ae. Distributed k-mutual exclusion problem and k-coteries. In Proceedings of the 2nd International Symposium on Algorithms (LNCS 557), pages 22-31, 1991.

[KFYA93] H. Kakugawa, S. Fujita, M. Yamashita, and T. Ae. Availability of k-coterie. IEEE Transactions on Computers, 42 (5):553-558, 1993.

[KFYA95] H. Kakugawa, S. Fujita, M. Yamashita, and T. Ae. A distributed k-mutual exclusion algorithm using k-coterie. Information Processing Letters, in press.

[NM94] M.L. Neilsen and M. Mizuno. Nondominated k-coterie for multiple mutual exclusion. Information Processing Letters, 50(5):247-252, 1994.

[SS94] M. Signal and N.G. Shivaratri. Lamport's logical clocks. Advanced Concepts in Operating Systems, pages 100-104, 1994.

# APPENDIXES

# APPENDIX A

# HEADER FILE

## inet.h

```
/* Definitions for TCP and UDP client/server programs. */
#include        <stdio.h>
#include        <string.h>
#include        <unistd.h>
#include        <sys/types.h>
#include        <sys/socket.h>
#include        <sys/select.h>
#include        <netinet/in.h>
#include        <arpa/inet.h>
#include        <pwd.h>
#include        <netdb.h>
#include        <parallel/parallel.h>
#define SERV_HOST_ADDR  "139.78.113.1"  /* host addr for server */
```

## length.h

```
/* server accepted at most MAX_USER clients at one time */
#define     MAX_USER        20
#define     MAX_LICENSE     256
#define     MAX_LINE        80      /* maximum characters per line */
#define     MAX_CHAR        240     /* maximum characters transmitted */
#define     ADDR_LEN        40      /* max length of addresses */
#define     PORT_LEN        10      /* max length of ports */

/* semaphore keys and shared memory key */
#define         SEMKEY_VAL1     43986
#define         SEMKEY_VAL2     43987
#define         SEMKEY_VAL3     43988
#define         SEMKEY_VAL4     43989
#define         SEMKEY_VAL5     43990
#define         SEMKEY_VAL6     43991

/* return value of child processes from fork call */
#define     CHILD       0
#define     NO          0
#define     YES         1
#define     OK          2
#define     WAIT        3
#define     RELEASE     4
#define     QUERY       5
#define     RELINQUISH  6
#define     D           1
```

## struct.h

```
#include "length.h"

/* table keeps information of local license managers connected to remote
 * license controllers (or information of users connected to local license
 * mangers). The table is implemented by using linked list
 */
struct table {
    int   node;  /* priority number of LLM or users:1=highest,2=high, ...   */
                 /* LLM: ordered by LLM address                             */
                 /* user: ordered by first come will get higher priority    */
    int   new_fd;          /* file descriptor */
```

```
    char   addr[ADDR_LEN];          /* LLM address */
    struct table *tptr;             /* pointer to next node */
};

/* table keeps information of remote license controllers connected to a
 * local license manager.  This table is implemented by using linked list
 */
struct serv_table  {
    int    node;   /* ID = server number (in ascending order of address) */
    int    new_fd;                 /*servers' socket fd */
    char   addr[ADDR_LEN];         /* servers' addresses */
    int    port;                   /* servers' ports */
    struct serv_table *sptr;       /* pointer to next node */
};

/* list of nodes associated with k-coteries = structure of each quorum
 */
struct llist {
  int    node;              /* server number (ID.) in each quorum */
  struct llist *ptr;        /* pointer to next node in the quorum */
};

/* k-coteries constructed by local license managers */
struct kcot  {
   int    kcot_num;         /* the order (number) of each quorum */
   struct llist *q;         /* pointer pointed to each quorum */
   int size;                /* size of each quorum */
   struct kcot *kptr;       /* pointer to next quorum */
};

/* cot_chosen is the structure that kept :
 * kcot_num  = the numbers of quorums in k-coteries
 * serv_selected = the server in this kcot_num quorum is
 *               selected to service users
 * *next = pointer to next quorum selected
 */
struct cot_chosen  {
   int kcot_num;
   int serv_selected;
   struct cot_chosen *next;
};

/* active_user_server is the structure that kept:
 * user_node = the number(ID.) of user who got license.
 * serv_node = the number (ID.) of server that serviced this user (user_node)
 * *usptr    = pointer to next node
 */
struct active_user_server  {
   int user_node;
   int serv_node;
   struct active_user_server *usptr;
};
```

```
/* timestamp (t, p) sent by local license managers to remote license
 * controller or by users to local license managers.
 * t = log_time, p = node
 */
struct timestamp {
   /* logical local time at which a process initiates a request */
   unsigned long log_time;
   int node;                    /* unique process priority or ID. */
   struct timestamp *tsptr;     /* pointer to next timestamp */
};
```

# APPENDIX B

# PROGRAM GENERATED K-COTERIE

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "struct.h"
extern void put_on_cotsel_asc(struct cot_chosen **, int, int);
extern void print_cotsel_asc(struct cot_chosen *);
struct kcot* form_kcot1(int, struct llist *);
struct kcot* form_kcot_w_eqvote(struct llist *);
struct kcot* form_kcot2(int, struct llist *, struct llist *, int);
struct kcot* form_kcot2_mix(int, struct llist *, struct llist *, int);
struct llist* create_temp_set(int, int, struct llist **);
struct kcot* select_quorum(struct kcot*, struct timestamp *, struct timestamp *, struct
cot_chosen **);
void free_llist(struct llist **);
void free_kcot(struct kcot **);
void upd_quorum_num(struct kcot *);
struct kcot* get_quorum_by_kcot_num(struct kcot *, int);
int chk_node_in_quorum(struct kcot *, int);

/*----------------------------------------------------------------
 * create_coterie: create k-coteries from the input data.
 * N   = the number of license servers,
 * k   = the number of licenses,
 * tab = set of license servers,
 * size = k-coterie's size (total number of quorums)
 * this function return the k-coteries created, and size of k-coterie
 *----------------------------------------------------------------*/
struct kcot* create_coterie(int N, int k, struct serv_table *tab, int *size) {
   struct serv_table *run_tab;
   struct kcot  *cot; /* k-coteries */
   struct llist *M;  /* selected set M = subset */
   struct llist *R;  /* subset of S => M intersect R = empty set */
   struct llist *head_M;  /* head of set M */
   struct llist *head_R;  /* head of set R */
   struct llist *run_M;  /* running pointer of set M */
   struct llist *run_R;  /* running pointer of set R */
   double maj, tot;
   int   max_vote; /* maximum value of vote for set S */
   int   max_member;
   int   i;
   int   count_R;  /* total number of set R */
   cot    = NULL;
   M      = NULL;
   head_M = NULL;
   run_M  = NULL;
   R      = NULL;
   head_R = NULL;
   run_R  = NULL;
   maj = 0;
   tot = 0;
   max_member = 0;

   run_tab = tab;
   if (run_tab == NULL)
     return cot;
   /* calculate MAJ, TOT according to theory of vote assignment */
   maj = ceil((double)(N+1)/(double)(k+1));
   tot = ((k+1) * maj) - 1;
```

```
if (N == tot)  {  /* every node has the same number of votes: one vote */
   max_vote = 1;
   max_member = N;
}
else if (N < tot)  {  /* some nodes have 2 votes, the others have 1 vote */
   max_vote = 2;
   max_member = tot - N;
}

for (i = 0; i < max_member; i++)  {  /* create selected set M */
   if (run_tab != NULL)  {
      M = (struct llist *) shmalloc(sizeof(struct llist));
      if (M == NULL)  {
         printf("Could not Allocate Memory Space!!\n");
         if (head_M != NULL)  {
            run_M  = NULL;
            free_llist(&head_M);
         }
         if (head_R != NULL)  {
            run_R  = NULL;
            free_llist(&head_R);
         }
         cot = NULL;
         return cot;
      }

      M->ptr  = NULL;
       M->node = run_tab->node;

      if (head_M == NULL)  {
         head_M = M;
         run_M  = M;
      }
      else  {
         run_M->ptr = M;
         run_M      = run_M->ptr;
      }
       run_tab = run_tab->sptr;
   }
}

count_R = 0;
while (run_tab != NULL)  {    /* create the rest of set = (tab - M) */
   R = (struct llist *) shmalloc(sizeof(struct llist));
   if (R == NULL)  {
      printf("Could not Allocate Memory Space!!\n");
      if (head_M != NULL)  {
         run_M  = NULL;
         free_llist(&head_M);
      }
      if (head_R != NULL)  {
         run_R  = NULL;
         free_llist(&head_R);
      }
      cot = NULL;
      return cot;
   }
```

```
    R->ptr  = NULL;
    R->node = run_tab->node;

    if (head_R == NULL)  {
      head_R = R;
      run_R  = R;
    }
    else  {
      run_R->ptr = R;
      run_R      = run_R->ptr;
    }
    count_R++;
    run_tab = run_tab->sptr;
  }

  if (max_vote == 1)  {
    cot = form_kcot1(maj, head_M);
  }
  else  {
    cot = form_kcot2(maj, head_M, head_R, count_R);
  }

  upd_quorum_num(cot);
  free_llist(&head_M);
  free_llist(&head_R);
  M       = NULL;
  head_M  = NULL;
  run_M   = NULL;
  R       = NULL;
  head_R  = NULL;
  run_R   = NULL;
  run_tab = NULL;
  return(cot);
}

/*-----------------------------------------------------------------
 * form_kcot1: form a k-coterie from list.  Each member in the list has 1 vote
 * head_M   = input list
 * tot_vote = total votes of each quorum
 * eg. head_M = 1, 2, 3, 4, 5, 6
 *    if tot_vote = 4
 *    then k-coterie = (1,2,3,4),(1,2,3,5),(1,2,3,6),(1,2,4,5),(1,2,4,6),
 *                   (1,2,5,6),(1,3,4,5),(1,3,4,6),(1,3,5,6),(1,4,5,6),
 *                   (2,3,4,5),(2,3,4,6),(2,3,5,6),(3,4,5,6)
 *-----------------------------------------------------------------------*/
struct kcot* form_kcot1(int tot_vote, struct llist *head_M)  {
  struct llist    *run_M;
  struct llist    *cur_M;
  struct llist    *newset;
  struct llist    *run_newset;
  struct llist    *head_newset;
  struct llist    *pre_newset;
  struct llist    *list;
  struct llist    *run_list;
  struct llist    *head_list;
  struct kcot     *cot;
  struct kcot     *run_cot;
  struct kcot     *head_cot;
```

```
int                tot, num, temp_node, loop, flag, tempset_num, size;
cot     = NULL;
head_cot = NULL;
run_cot  = NULL;
head_newset = NULL;
pre_newset  = NULL;
run_newset  = NULL;
tot = 0;
num = 0;

cur_M = head_M;
if ((cur_M == NULL) || (tot_vote < 1))
  return cot;
if (tot_vote == 1)  {  /* total vote in each quorum = 1 */
   cot = form_kcot_w_eqvote(head_M);
  return cot;
}
while (cur_M != NULL)  {
  run_M = cur_M;
  tot++;
  head_newset = NULL;

  /* create temporary set with its size(vote) = (tot_vote -1)
   * eg. tot_vote = 4.  If quorum = (1,2,3,4),(1,2,3,5),(1,2,3,6)
   *                  then temporary set = (1,2,3)
   */
  head_newset = create_temp_set(tot, tot_vote, &run_M);
  if (head_newset != NULL)  {
     tempset_num = tot_vote - tot;
     size = tempset_num;
  }
  else  {
     cur_M = cur_M->ptr;
     tot = 0;
     continue;
  }

  while (run_M != NULL)  {
     while (run_M != NULL)  {
        /* create k-coterie */
        cot = (struct kcot *) shmalloc(sizeof(struct kcot));
        if (cot == NULL)  {
           printf("Could not Allocate Memory Space!!\n");
           if (head_cot != NULL)  {
              run_cot = NULL;
              free_kcot(&head_cot);
              head_cot = NULL;
           }
           if (head_newset != NULL)  {
              run_newset = NULL;
              pre_newset = NULL;
              free_llist(&head_newset);
           }
           return head_cot;
        }

        cot->kptr = NULL;
        cot->size = 0;
```

```
if (head_cot == NULL)  {
   head_cot  = cot;
   run_cot   = cot;
}
else  {
   run_cot->kptr = cot;
   run_cot      = run_cot->kptr;
}

num       = 0;
list      = NULL;
run_list  = NULL;
head_list = NULL;
run_newset = head_newset;
while (run_newset != NULL)  {
    /* add nodes from temporary set to be members of a quorum in *
    * k-coteries                                          */
   list = (struct llist *) shmalloc(sizeof(struct llist));
   if (list == NULL)  {
     printf("Could not Allocate Memory Space!!\n");
     if (head_cot != NULL)  {
        run_cot = NULL;
        free_kcot(&head_cot);
        head_cot = NULL;
     }
     if (head_newset != NULL)  {
        run_newset = NULL;
        pre_newset = NULL;
        free_llist(&head_newset);
     }
     return head_cot;
   }

   list->ptr  = NULL;
   list->node = run_newset->node;
   if (head_list == NULL)  {
     run_cot->q      = list;
     head_list   = list;
     run_list    = list;
   }
   else  {
     run_list->ptr = list;
     run_list      = run_list->ptr;
   }
   num++;
   run_newset = run_newset->ptr;
}

/* add one more node in a quorum so the quorum's vote = tot_vote */
list = (struct llist *) shmalloc(sizeof(struct llist));
if (list == NULL)  {
   printf("Could not Allocate Memory Space!!\n");
   if (head_cot != NULL)  {
     run_cot = NULL;
     free_kcot(&head_cot);
     head_cot = NULL;
   }
```

```
      if (head_newset != NULL) {
        run_newset = NULL;
        pre_newset = NULL;
        free_llist(&head_newset);
      }
      return head_cot;
    }

    list->ptr  = NULL;
    list->node = run_M->node;
    if (run_list != NULL) {
      run_list->ptr = list;
      run_list      = run_list->ptr;
    }
    else {
      if (run_cot->q == NULL)
        run_cot->q = list;
      else {
        free_llist(&list);
        printf("Error in constructing k-coteries\n");
        if (head_cot != NULL) {
          run_cot = NULL;
          free_kcot(&head_cot);
          head_cot = NULL;
        }
        if (head_newset != NULL) {
          run_newset = NULL;
          pre_newset = NULL;
          free_llist(&head_newset);
        }
        return head_cot;
      }
    }
    num++;
    run_cot->size = num;
    run_M         = run_M->ptr;
  }

  /* from temporary set, delete the last member, and add the new member which gets
   * from input list (run_M)
   * eg. run_M = (1,2,3,4,5,6)
   * before:  temporary set = (1,2,3)
   * after doing the following code: temporary set = (1,2,4)
   */
  flag = 0;
  loop = 1;
  while (loop == 1) {
    run_newset = head_newset;
    pre_newset = head_newset;
    run_M      = head_M;
    while ((run_newset != NULL) && ((run_newset->ptr) != NULL)) {
      if (run_newset != head_newset) {
        pre_newset = run_newset;
      }
      run_newset = run_newset->ptr;
      run_M      = run_M->ptr;
    }
```

```
if ((pre_newset != NULL) && (run_newset != head_newset)) {
    temp_node = run_newset->node;
   pre_newset->ptr = NULL;
    shfree((char *) run_newset);
   size--;
   run_newset = pre_newset;
    while((run_M->node < temp_node) && (run_M != NULL))
      run_M = run_M->ptr;

   run_M      = run_M->ptr;
    while ((run_M != NULL) && (run_M->ptr != NULL) &&
         (size < tempset_num)) {
      newset = (struct llist *) shmalloc(sizeof(struct llist));
      if (newset == NULL) {
         printf("Could not Allocate Memory Space!!\n");
         if (head_cot != NULL) {
            run_cot = NULL;
             free_kcot(&head_cot);
            head_cot = NULL;
         }
         if (head_newset != NULL) {
            run_newset = NULL;
            pre_newset = NULL;
            free_llist(&head_newset);
         }
         return head_cot;
      }

      newset->ptr  = NULL;
       newset->node = run_M->node;
      run_newset->ptr = newset;
      run_newset     = run_newset->ptr;
      run_M       = run_M->ptr;
     size++;
   }
   if (size < tempset_num) {
      if (run_newset != head_newset)  {
        continue;
      }
      else {
         run_M = NULL;
      }
   }
}
else {
   pre_newset = NULL;
   run_newset = NULL;
   if (head_newset != NULL) {
      if (head_newset->ptr != NULL) {
         printf("Error at head of list \n");
         free_llist(&head_newset);
      }
      else {
          shfree((char *) head_newset);
      }
      head_newset = NULL;
   }
   flag = 1;
```

```
          }
          loop = 0;
        } /* while (loop == 1) */
        if (flag == 1)
          break;
      }
      if (head_newset != NULL)
        free_llist(&head_newset);
      tot = 0;
      cur_M = cur_M->ptr;
    }
    if (head_newset != NULL)
      free_llist(&head_newset);
    return(head_cot);
}

/*---------------------------------------------------------------------
 * form_kcot_w_eqvote: create kcoterie (each quorum has only 1
 *                     member(node) (each node has 1 vote (or 2 votes) for 1 license
 *                     (or 2 licenses)
 * head_M = the head of list M
 *---------------------------------------------------------------------*/
struct kcot* form_kcot_w_eqvote(struct llist *head_M) {
    struct llist   *run_M;
    struct llist   *list;
    struct kcot    *head_cot;
    struct kcot    *cot;
    struct kcot    *run_cot;
    run_M    = head_M;
    head_cot = NULL;

    while (run_M != NULL) {
      /* create k-coterie */
      cot = (struct kcot *) shmalloc(sizeof(struct kcot));
      if (cot == NULL) {
        printf("Could not Allocate Memory Space!!\n");
        if (head_cot != NULL) {
          run_cot = NULL;
          free_kcot(&head_cot);
          head_cot = NULL;
          cot      = NULL;
        }
        return cot;
      }

      cot->kptr = NULL;
      cot->size = 1;
      if (head_cot == NULL) {
        head_cot  = cot;
        run_cot   = cot;
      }
      else {
        run_cot->kptr = cot;
        run_cot       = run_cot->kptr;
      }
```

```
    list = (struct llist *) shmalloc(sizeof(struct llist));
    if (list == NULL)  {
       printf("Could not Allocate Memory Space!!\n");
       if (head_cot != NULL)  {
          run_cot = NULL;
          free_kcot(&head_cot);
          head_cot = NULL;
          cot      = NULL;
       }
       return cot;
    }

    list->ptr  = NULL;
    list->node = run_M->node;
    cot->q     = list;

    run_M      = run_M->ptr;
  }
  return(head_cot);
}

/*-----------------------------------------------------
 * form_kcot2: form kcoterie from two lists (M and R)
 * maj      = total votes in each quorum
 * head_M   = the head of list M
 * head_R   = the head of list R
 * count_R  = total number of members in list R
 *-----------------------------------------------------*/
struct kcot* form_kcot2(int maj, struct llist *head_M, struct llist *head_R, int count_R)  {
  struct kcot     *cot;
  struct kcot     *head_cot;
  struct kcot     *run_cot;
  int    tot_vote, temp_totvote;
  cot          = NULL;
  head_cot     = NULL;
  run_cot      = NULL;
  tot_vote     = 0;
  temp_totvote = 0;
  if ((count_R == 0) && (head_R != NULL))  {
     printf("count_R must be greater than zero if set R != NULL \n");
     head_cot = NULL;
     return head_cot;
  }

  if (head_M != NULL)  {
     if (maj == 2)  {
        head_cot = form_kcot_w_eqvote(head_M);
     }
     else if (maj > 2)  {
        if ((maj % 2) == 0)
          tot_vote = maj;
        else
          tot_vote = maj + 1;

        temp_totvote = tot_vote / 2;
        head_cot = form_kcot1(temp_totvote, head_M);
     }
```

```
    /* go to the last quorum in k-coterie */
    if (head_cot != NULL) {
      run_cot = head_cot;
      while (run_cot->kptr != NULL)
        run_cot = run_cot->kptr;
    }
  }

  if (head_R != NULL) {
    if (maj == 2) {
      if (count_R > 1) {
        /* create the new coterie formed from list R only */
        cot = form_kcot1(maj, head_R);
      }
      else if (count_R == 1) {
        /* create the new coterie formed from two lists (M and R) */
        cot = form_kcot2_mix(maj, head_M, head_R, count_R);
      }

      /* connect the created coterie with the previous k-coterie */
      if (run_cot != NULL)
        run_cot->kptr = cot;
      else
        head_cot = cot;
    }
    else if (maj > 2) {
      /* create the new coterie formed from two lists (M and R) */
      cot = form_kcot2_mix(maj, head_M, head_R, count_R);

      /* connect the created coterie with the previous k-coterie */
      if (run_cot != NULL)
        run_cot->kptr = cot;
      else {
        head_cot = cot;
        run_cot  = cot;
      }

      /* go to the last quorum in k-coterie */
      if (run_cot != NULL) {
        while (run_cot->kptr != NULL)
          run_cot = run_cot->kptr;
      }

      /* create the new coterie formed from list R only */
      cot = form_kcot1(maj, head_R);

      /* connect the created coterie with the previous k-coterie */
      if (run_cot != NULL)
        run_cot->kptr = cot;
      else
        head_cot = cot;
    }
  }
  return head_cot;
}
```

```
/*-----------------------------------------------------------------------
 * form_kcot2_mix: form kcoterie from two lists (M and R) with mixing
 *                 members (from M and R)
 * maj      = total votes in each quorum
 * head_M   = the head of list M
 * head_R   = the head of list R
 * count_R  = the total members (nodes) on list R
 *-----------------------------------------------------------------------*/
struct kcot* form_kcot2_mix(int maj, struct llist *head_M, struct llist *head_R, int count_R)
{
   struct kcot   *cot;
   struct kcot   *run_cot;
   struct kcot   *head_cot;
   struct kcot   *last_cot;
   struct llist  *run_M;
   struct llist  *cur_M;
   struct llist  *newset;
   struct llist  *head_newset;
   struct llist  *run_newset;
   struct llist  *list;
   struct llist  *head_list;
   struct llist  *run_list;
   struct llist  *last_list;
   int           temp_div, temp_mod, tot, temp_tot, num, i, j;
   cot         = NULL;
   run_cot     = NULL;
   head_cot    = NULL;
   last_cot    = NULL;
   newset      = NULL;
   head_newset = NULL;
   run_newset  = NULL;
   last_list   = NULL;

   if ((head_M == NULL) || (head_R == NULL))
      return head_cot;

   if ((maj <= 2) && (count_R > 1))  {
      /* maj for this case must be greater than 2 */
      return head_cot;
   }

   temp_div = maj / 2;
   temp_mod = maj % 2;

   cur_M = head_M;
   while (cur_M != NULL)  {
      run_M = cur_M;
      tot = 0;
      for (i = 1; i <= temp_div; i++)  {
         if (head_newset != NULL)  {
            free_llist(&head_newset);
         }
         head_newset = NULL;
         run_newset = NULL;
         head_newset = create_temp_set(tot, i, &run_M);
         if (head_newset == NULL)  {
            continue;
         }
```

```
num = 0;
run_newset = head_newset;
while (run_newset != NULL)  {
  num++;
  run_newset = run_newset->ptr;
}

if (((num * 2) > maj) ||
    (((num * 2) == maj) && (count_R != 1)))  {
   printf("Error about votes per quorum!!\n");
   if (head_cot != NULL)  {
      run_cot = NULL;
      free_kcot(&head_cot);
      head_cot = NULL;
   }
   if (head_newset != NULL)  {
      run_newset = NULL;
      free_llist(&head_newset);
   }
   return head_cot;
}

if (((num * 2) == maj) && (count_R == 1))
   temp_tot = 1;
else
   temp_tot = maj - (num * 2);

cot = form_kcot1(temp_tot, head_R);
if (cot != NULL)  {
   if (head_cot == NULL)  {
      head_cot  = cot;
      run_cot   = cot;
   }
   else  {
      run_cot->kptr = cot;
      run_cot       = run_cot->kptr;
   }

   while (run_cot != NULL)  {
      run_list    = NULL;
      head_list   = NULL;
      last_list   = NULL;
      run_newset  = head_newset;
      while (run_newset != NULL)  {
         /* add nodes from temporary set to be members of a  *
          * quorum in k-coteries                              */
         list = (struct llist *) shmalloc(sizeof(struct llist));
         if (list == NULL)  {
            printf("Could not Allocate Memory Space!!\n");
            if (head_cot != NULL)  {
               run_cot = NULL;
               free_kcot(&head_cot);
               head_cot = NULL;
            }
            if (head_newset != NULL)  {
               run_newset = NULL;
               free_llist(&head_newset);
```

```
                }
                return head_cot;
            }

            list->ptr  = NULL;
            list->node = run_newset->node;
            if (head_list == NULL)  {
              head_list   = list;
              run_list    = list;
            }
            else  {
              run_list->ptr = list;
              run_list      = run_list->ptr;
            }
            if (run_newset->ptr == NULL)
              last_list = run_list;
            run_newset = run_newset->ptr;
          }

          if (last_list != NULL)  {
            last_list->ptr = run_cot->q;
            run_cot->q  = head_list;
            (run_cot->size)++;
          }
          if (run_cot->kptr == NULL)
            last_cot = run_cot;
          run_cot = run_cot->kptr;
        }  /* while */
        run_cot = last_cot;
      }  /* if */
    }  /* for */
    cur_M = cur_M->ptr;
  }  /* while */
  return head_cot;
}

/*------------------------------------------------------------
 * create_temp_set: create temporary set with
 *                  its size(vote) = (tot_vote - tot)
 * tot = start number ==> tot_vote - tot = size of temporary set
 * tot_vote = total vote in each quorum
 * run_M    = list that will be used to create temporary set
 *-----------------------------------------------------------*/
struct llist* create_temp_set(int tot, int tot_vote, struct llist **run_M)  {
  struct llist   *newset;
  struct llist   *head_newset;
  struct llist   *run_newset;

  newset      = NULL;
  head_newset = NULL;
  run_newset  = NULL;
  while ((tot < tot_vote) && (*run_M != NULL))  {
    /* create temporary set */
    newset = (struct llist *) shmalloc(sizeof(struct llist));
    if (newset == NULL)  {
      printf("Could not Allocate Memory Space!!\n");
      if (head_newset != NULL)  {
        run_newset = NULL;
```

```
        free_llist(&head_newset);
      }
      head_newset = NULL;
      return head_newset;
    }

    newset->ptr  = NULL;
    newset->node = (*run_M)->node;
    if (head_newset == NULL)  {
      head_newset = newset;
      run_newset  = newset;
    }
    else  {
      run_newset->ptr = newset;
      run_newset      = run_newset->ptr;
    }
    *run_M  =  (*run_M)->ptr;
    tot++;
  }
  if (tot < tot_vote)  {
    run_newset = NULL;
    free_llist(&head_newset);
    head_newset = NULL;
  }
  return head_newset;
}

/*--------------------------------------------------------------
 * free_llist: free memory space allocated for list (struct llist)
 * head = list that wants to be deallocated
 *--------------------------------------------------------------*/
void free_llist(struct llist **head)  {
  struct llist *del_ptr;

  del_ptr = *head;
  while (del_ptr != NULL)  {
     *head = (*head)->ptr;
    del_ptr->ptr = NULL;
    shfree((char *) del_ptr);
    del_ptr      = *head;
  }
  return;     /* At this point, head and del_ptr are NULL */
}

/*---------------------------------------------------------------
 * free_kcot: free memory space allocated for k-coterie(struct kcot)
 * head = k-coterie that wants to be deallocated
 *---------------------------------------------------------------*/
void free_kcot(struct kcot **head)  {
  struct kcot *del_ptr;
  struct llist *head_list;

  del_ptr = *head;
  while (del_ptr != NULL)  {
     *head = (*head)->kptr;
    del_ptr->kptr = NULL;
    if (del_ptr->q != NULL)  {
      head_list = del_ptr->q;
```

```
        del_ptr->q = NULL;
        free_llist(&head_list);
      }
      shfree((char *) del_ptr);
      del_ptr      = *head;
  }
  /* At this point, head aind del_ptr are NULL */
  return;
}

/*---------------------------------------------------------------
 * upd_quorum_num:  update kcot_num (quorum number of each quorum).
 *---------------------------------------------------------------*/
void upd_quorum_num(struct kcot *cot)  {
  struct kcot *run_cot;
  int num;

  num = 0;

  for (run_cot = cot; run_cot != NULL; run_cot = run_cot->kptr)  {
    num++;
     run_cot->kcot_num = num;
  }
  return;
}

/*-----------------------------------------------------------------
 * select_quorum: search for a quorum satisfied the following condition:
 *              quorum Q' which maximize | Q intersect YES| from quorms
 *             in k-coterie C not intersecting with NOTNOW.
 * cot = k-coterie formed.
 * yes = YES table kept the RLCs sent OK messages to LLM.
 * notnow = NOTNOW table kept the RLCs sent WAIT messages to LLM.
 * visited_cot = quorums has been visited.
 *-----------------------------------------------------------------*/
struct kcot* select_quorum(struct kcot* cot, struct timestamp *yes, struct timestamp
*notnow, struct cot_chosen **visited_cot)  {
  int selected_no;
  struct cot_chosen     *run_sel_cot;
  struct kcot           *tail;
  struct llist          *run_list;
  struct timestamp      *run_yes;
  struct timestamp      *run_notnow;
  int    in_notnow;  /* 0 = not in notnow; 1 = in notnow queue */
  int    max_intersect;
  int    max_kcot_num;
  int    intersect_no;

  max_intersect = -1;  /* set max_intersect to be the small number */
  max_kcot_num  = 0;
  tail = cot;
  if (*visited_cot == NULL)  {  /* select quorum for the first time */
    put_on_cotsel_asc(visited_cot, tail->kcot_num, 0);
    return tail;
  }
```

```
while (tail != NULL) {
   selected_no = 0;
   run_sel_cot = *visited_cot;
   while (run_sel_cot != NULL) {
      /* check if there is any quorum already selected */
      if (run_sel_cot->kcot_num == tail->kcot_num) {
        selected_no = 1;
        break;
     }
      else if (run_sel_cot->kcot_num > tail->kcot_num) {
        /* kcot_num in visited_cot is in the ascending order.
         * This mean, tail (the current quorum) has not been selected.
        */
        break;
     }
      run_sel_cot = run_sel_cot->next;
   }

   if (selected_no == 1) { /* this quorum has been selected */
     tail = tail->kptr;
     continue;
   }
   else { /* if (selected_no == 0) => this quorum hasn't been selected */
     in_notnow = 0;
     if (notnow != NULL) { /* there are some nodes in notnow queue */
       run_list = tail->q;
        /* compare each node in the quorum with each element in notnow */
       while (run_list != NULL) {
          run_notnow = notnow;
          while (run_notnow != NULL) {
              if (run_list->node == run_notnow->node) {
               in_notnow = 1;
               break;
            }
             run_notnow = run_notnow->tsptr;
          } /* while run_notnow */

          if (in_notnow == 1) { /* a node in quorum is in notnow queue */
            break;
          }

          run_list = run_list->ptr;
       } /* while run_list */
       if (in_notnow == 1) {
          /* a node in the current quorum is in notnow */
          tail = tail->kptr;
          continue; /* check a new quorum in k-coteries */
       }
     } /* if notnow */

     /* the set that is not intersect with notnow will be
      * checked | Q intersect YES | = maximization
     */

     /* check size of set (quorum intersect YES) */
     intersect_no = 0;
     run_list = tail->q;
     while (run_list != NULL) {
```

```
          run_yes = yes;
          while (run_yes != NULL)  {
             if (run_list->node == run_yes->node)  {
               intersect_no++;
               break;
            }
            run_yes = run_yes->tsptr;
          }
          run_list = run_list->ptr;
        }

        /* keep the maximize (quorum intersect YES) */
        if (intersect_no > max_intersect)  {
           max_intersect = intersect_no;
           max_kcot_num  = tail->kcot_num;
        }
     }  /* else of if selected_no */
     tail = tail->kptr;
   }

   if (max_kcot_num != 0)  {
     /* this is a quorum satifying the condition:
      * maximize |Q intersect YES| from quorums in the k-coteries not
      * intersecting with notnow */

     tail = cot;
     while (tail != NULL)  {
        if (tail->kcot_num == max_kcot_num)  {
          break;
       }
       tail = tail->kptr;
     }

     if (tail != NULL)  {
        put_on_cotsel_asc(visited_cot, tail->kcot_num, 0);
     }
     else  {
        printf("there is an error in select_quorum\n");
       exit(1);
     }
   }
   else  {
     /* could not find a quorum satisfying the condition */
     tail = NULL;
   }
   return tail;
}

/*-------------------------------------------------------------------
 * get_quorum_by_kcot_num: get a quorum by using kcot_num as an index
 * cot = k-coterie formed.
 * kcot_num = the number of the coterie wanted to get.
 *-------------------------------------------------------------------*/
struct kcot* get_quorum_by_kcot_num(struct kcot* cot, int kcot_num)  {
   struct kcot *Cptr;
   struct kcot *temp;
   temp = NULL;
   Cptr = cot;
```

```
  /* check if cot != NULL */
  if (Cptr == NULL)
    return temp;

  /* search for the element whose values are equal to input arguments */
  while ((Cptr != NULL) && (Cptr->kcot_num != kcot_num)) {
    Cptr = Cptr->kptr;
  }

  if (Cptr == NULL)
    return temp;
  else
    return Cptr;
}

/*--------------------------------------------------------------------------
 * chk_node_in_quorum: check if the input node (node) is in a quorum (in_cot).
 *--------------------------------------------------------------------------*/
int chk_node_in_quorum(struct kcot *in_cot, int node)  {
  struct llist *run_list;
  int       temp;
  temp = 0;

  /* check if cot != NULL */
  if (in_cot == NULL)
    return temp;

  /* search for the element whose values are equal to input arguments */
  for (run_list = in_cot->q; run_list != NULL; run_list = run_list->ptr)  {
    if (run_list->node == node)  {
      temp = 1;
      break;
    }
  }
  return temp;
}
```

## APPENDIX C

## DISTRIBUTED LICENSE SERVER

**tcpserv1.c**

```
#include      "inet.h"
#include      "length.h"
#include      "struct.h"

extern int get_port_from_file(char *, char **);
extern struct table* get_tab_from_node(struct table *, int);
extern struct timestamp *get_data_from_tst_node(struct timestamp *, int);
extern int serv_connect(int);
extern void put_on_tab(struct table **, int, char *);
extern void put_on_tab_asc(struct table **, int, char *);
extern void put_on_tst_asc(struct timestamp **, int, int);
extern void upd_tab_node(struct table *);
extern ulong adjust_log_time(ulong, ulong);
extern void print_tab(struct table *);
extern void del_from_tab(struct table **, int);
extern void del_from_tst(struct timestamp **, int log_time, int node);
extern void del_from_tst_by_node(struct timestamp **, int node);
void exit_rtn(struct table **, int);
char* chk_lock(struct timestamp **, struct timestamp **, int, int);
struct timestamp* min_timestamp(struct timestamp *, struct timestamp *);
int chk_next_request(struct timestamp **, struct timestamp **);
void query_server(struct table *, int, ulong *, int *);

main(int argc, char *argv)
{
  /* keep the timestamp of the user who got the permission */
  struct timestamp *perm;

  /* keep timestamps of the users who asked the permission */
  struct timestamp *queue;

  char                mesg[MAX_CHAR];
  char                in_str[MAX_CHAR];
  char                in_str1[MAX_CHAR];
  char                in_str2[MAX_CHAR];
  char                out_str[MAX_CHAR];
  char             c_addr[ADDR_LEN];
  char             *own_addr;
  int             own_port;
  int                 sockfd, newsockfd, max_desc, clilen;
  struct sockaddr_in  cli_addr;
  struct timeval      wait_time;
  fd_set              readset;
  struct table        *tab;
  struct table        *tail;
  struct table        *temp_tab;

  struct table        *run_ptr;
  struct timestamp   *temp_tst;

  int           pid;
  ulong            log_time; /*logical time of remote license controller*/
  ulong           in_logtime;
  ulong           temp_logtime;
  int            temp_node;
  char            time_str[MAX_LINE];
  int            temp_fd;
```

```
int             terminate_flag;
int             answer;
int                     i, j;
int             clnt_num;
int             rec_msg;
int             snd_msg;

clnt_num = 0;
rec_msg = 0;
snd_msg = 0;

if (argc < 2)  {
   printf("Usage:  tcpserv <host_addr[version]> \n");
   exit(0);
}

/* get port number from file */
own_addr = (char *) malloc(sizeof(char[ADDR_LEN]));
own_port = get_port_from_file(argv[1], &own_addr);

/* open TCP socket and bind local address so client can send to the server */
sockfd = serv_connect(own_port);

wait_time.tv_sec=5;
wait_time.tv_usec=0;        /* Wait time is 10 seconds */

if (listen(sockfd, MAX_USER) < 0)  {
   err_sys("listen error");
}
tab   = NULL;
perm  = NULL;
queue = NULL;
log_time = 0;               /* which will initiate to 0 */
terminate_flag = 0;

for ( ; ; )  {
  /*
   * Wait for a connection from a client process.
   */
   FD_ZERO(&readset);
   FD_SET(sockfd, &readset);  /* set fd of server socket */

  max_desc = sockfd;
   if (tab == NULL)    {
      if (terminate_flag == 1)  {
        /* if all LLMs are terminated, it will terminate, as well */
        break;
      }
   }
  else  {
     for (tail = tab; tail != NULL; tail = tail->tptr)  {
        FD_SET(tail->new_fd, &readset);
        if (max_desc < tail->new_fd)  {
           max_desc = tail->new_fd;
        }
     }
  }
```

```
/* see if any descriptors are ready to be read, check at most 5 sec. */
if ((j = select(max_desc+1,&readset,NULL,NULL,&wait_time)) > 0) {
  log_time += D;

  /* check if a client asks for connection */
  if (FD_ISSET(sockfd, &readset)) {
    clilen   = sizeof(cli_addr);
     newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
    if (newsockfd > 0)  {
      strcpy(in_str, "");
      read(newsockfd, &in_str, MAX_CHAR);
      rec_msg++;
      clnt_num++;
      strcpy(mesg, in_str);
      strcpy(in_str1, strtok(in_str, " \n"));
      strcpy(in_str2, strtok(NULL, "\n"));
       if ((strcmp(in_str1,"/addr") == 0) ||
          (strcmp(in_str1,"/addr_user") == 0))
         strcpy(c_addr, in_str2);
      else
         strcpy(c_addr, "");

       if (strcmp(in_str1,"/addr_user") == 0)  {
         /* put on table */
          put_on_tab(&tab, newsockfd, c_addr);
      }
       else  { /* if (strcmp(in_str1,"/addr") == 0) */
          /* put on table in ascending order */
          put_on_tab_asc(&tab, newsockfd, c_addr);
          /* update node (local license managers' priority) according
           * to ascending order of addresses */
          upd_tab_node(tab);
      }

    }
  }

    /* check to see if any client wants service */
    tail = tab;
  while (tail != NULL)  {
    strcpy(in_str, "");
     if (FD_ISSET(tail->new_fd, &readset)) {
       read(tail->new_fd, &in_str, MAX_CHAR);
       rec_msg++;
       strcpy(mesg, in_str);
       strcpy(in_str1, strtok(in_str, " \n"));
       strcpy(in_str2, strtok(NULL, "\n"));
       if (strcmp(in_str1, "/lock") == 0)  {
          /* logical clock is incremented between any two successive
           * events
          */
          in_logtime = atoi(in_str2);
           log_time = adjust_log_time(log_time, in_logtime);
           strcpy(out_str, chk_lock(&perm, &queue, tail->node, log_time));
          strcpy(mesg, out_str);
          itoa(log_time, time_str);
          strcat(out_str, time_str);
         strcat(out_str, "\n");
```

```
    write(tail->new_fd, &out_str, sizeof(out_str));
    snd_msg++;

    if (strcmp(mesg, "/QUERYING\n") == 0)  {
       if (perm->node != tail->node)  {
         /* in the case tha license server sends QUERY to
          * another LLM to see if user is using services
          * from this server */
         /* if perm->node == tail->node, do nothing because
          * LLM itself has already locked this server from
          * the previous request, then this request has to
          * wait until itself releases lock */
          query_server(tab, perm->node, &log_time, &snd_msg);
       }
    }
  }
  else if(strcmp(in_str1, "/release_lock") == 0)  {
     in_logtime = atoi(in_str2);
     log_time = adjust_log_time(log_time, in_logtime);

    if (perm == NULL)  {
       printf("<tcpserv> Error1 in /release_lock license server\n");
    }
    else  {
       /* there is a request in perm (a LLM locked this RLC) */
       if (perm->node == tail->node)  {
          del_from_tst(&perm, perm->log_time, perm->node);

          answer = chk_next_request(&perm, &queue);
        if (answer == YES)  {  /* there is a request in queue */
           if (perm == NULL)  {
              printf("<tcpserv> Error2 in /release_lock chk_next_request\n");
           }
           else  {
              temp_tab = get_tab_from_node(tab, perm->node);
              if (temp_tab == NULL)  {
                 printf("<tcpserv> Error3 in /release_lock get_tab_from_node\n");
              }
              else  {
                 strcpy(out_str, "/OK\n");
                 itoa(log_time, time_str);
                 strcat(out_str, time_str);

                 strcat(out_str, "\n");

                 write(temp_tab->new_fd, &out_str, sizeof(out_str));
                 snd_msg++;
              }
           }
        }
       }
       else  {
          printf("<tcpserv> Error4 in /release_lock license server\n");
       }
    }
  }
  else if(strcmp(in_str1, "/release_wait") == 0)  {
     in_logtime = atoi(in_str2);
```

```
    log_time = adjust_log_time(log_time, in_logtime);

    if (queue == NULL)  {  /* no record in queue */
      /* check if this record was moved to perm */
      if (perm == NULL)  {
        printf("<tcpserv> Error1 in /release_wait license server\n");
      }
      else  {  /* queue == NULL AND perm != NULL */
        if (perm->node != tail->node)  {
          printf("<tcpserv> Error2 in /release_wait license server\n");
        }
      }  /* else of if (perm) */
    }  /* if (queue) */
    else  {   /* queue != NULL */
      temp_tst = get_data_from_tst_node(queue, tail->node);
      if (temp_tst != NULL)  {
        del_from_tst_by_node(&queue, temp_tst->node);
      }
    }
}
else if(strcmp(in_str1, "/terminate") == 0)  {
    terminate_flag = 1;
    temp_fd = tail->new_fd;
    tail = tail->tptr;
    exit_rtn(&tab, temp_fd);
    continue;
}
else if(strcmp(in_str1, "/run_software") == 0)  {
    /* from users only */
    in_logtime = atoi(in_str2);
    log_time = adjust_log_time(log_time, in_logtime);

    strcpy(out_str, "Start-Running-Software\n");
    write(tail->new_fd, &out_str, sizeof(out_str));
    snd_msg++;
    sleep(1);

    log_time += D;

    strcpy(out_str, "Finish-Running-Software\n");
    itoa(log_time, time_str);
    strcat(out_str, time_str);
    strcat(out_str, "\n");
    write(tail->new_fd, &out_str, sizeof(out_str));
    snd_msg++;
}
else if(strcmp(in_str1, "/exit") == 0)  {
    /* from users only */
    in_logtime = atoi(in_str2);
    log_time = adjust_log_time(log_time, in_logtime);
    temp_fd = tail->new_fd;
    tail = tail->tptr;
    exit_rtn(&tab, temp_fd);
    continue;
}
else if(strcmp(in_str1, "/relinquish") == 0)  {
    in_logtime = atoi(in_str2);
    log_time = adjust_log_time(log_time, in_logtime);
```

```
            if (perm == NULL)  {
               printf("<tcpserv> /relinquish error at perm == NULL\n");
            }
            else  {
               if (perm->node != tail->node)  {
                  printf("<tcpserv> /relinquish error at perm->node != tail-node\n");
               }
               else  {
                  temp_logtime = perm->log_time;
                  temp_node    = perm->node;
                  del_from_tst(&perm, temp_logtime, temp_node);
                  mv_to_tst(&queue, &perm, queue->log_time, queue->node);
                  put_on_tst_asc(&queue, temp_logtime, temp_node);

                  /* send /OK to perm->node */
                  temp_tab = get_tab_from_node(tab, perm->node);
                  if (temp_tab == NULL)  {
                     printf("<tcpserv> Error /relinquish  get_tab_from_node\n");
                  }
                  else  {
                     strcpy(out_str, "/OK\n");
                     itoa(log_time, time_str);
                     strcat(out_str, time_str);
                     strcat(out_str, "\n");

                     write(temp_tab->new_fd, &out_str, sizeof(out_str));
                     snd_msg++;
                  }
               }
            }
         }
         else {
            printf("<tcpserv> message: ");
            write(1, &mesg, strlen(mesg));  /* 1 = stdout */
         }

      }  /* if FD_ISSET */
         tail = tail->tptr;
    }  /* while */
  }  /* if of select */
}  /* for */
free(own_addr);
if (tab != NULL)  {
   printf("<tcpserv> Terminate but tab != NULL\n");
   free_tab(&tab);
}
free_tst_tab(&perm);
free_tst_tab(&queue);
printf("number of LLMs/Users connecting to this RLC = %d\n", clnt_num);
printf("number of messages receiving from LLMs/Users = %d\n", rec_msg);
printf("number of messages sending to LLMs/Users = %d\n", snd_msg);
return;
}
```

```
/*-----------------------------------------------------------------------
 * exit_rtn: delete a user asking to exit) from table (tab) and close the fd
 *          of that user.
 * tab = table kept the information of all users
 * fd  = fd of the user asking to exit or disconnect from LLM
 *-----------------------------------------------------------------------*/
void exit_rtn(struct table **tab, int fd)  {
  del_from_tab(tab, fd);
  close(fd);
  return;
}

/*-----------------------------------------------------------------------
 * chk_lock: check if the RLC has been locked (PERM is not empty if it is
 *          locked).
 *          If PERM is empty, add request to PERM and send OK back to LLM.
 *          Otherwise, add request to QUEUE and check if this request is
 *                   the highest priority request (the smallest timestamp).
 *                   If no, send WAIT back to LLM.
 *                   Otherwise, send QUERYING to LLM kept in PERM.
 * perm  = PERM table kept the LLM locking this RLC
 * queue = QUEUE table kept requests sent from LLMs
 * node  = node number of LLM
 * log_time = timestamp of LLM
 *-----------------------------------------------------------------------*/
char* chk_lock(struct timestamp **perm, struct timestamp **queue, int node, int log_time)
{
  struct timestamp *temp_tst;
  struct timestamp *in_tst;
  char node_str[MAX_LINE];
  char answer[MAX_LINE];

  /* PERM is empty so add request (log_time, node) to PERM, and send OK */
  if (*perm == NULL)  {
    put_on_tst_asc(perm, log_time, node);
   strcpy(answer, "/OK\n");
    return(answer);
  }

  /* PERM is not empty */
  /* temp_tst = (struct timestamp *) malloc(sizeof(struct timestamp *)); */

  in_tst   = (struct timestamp *) malloc(sizeof(struct timestamp));
  /* put new request in queue and check the priority of new request */
  put_on_tst_asc(queue, log_time, node);
  if (((*queue)->log_time == log_time)  && ((*queue)->node == node))  {
  /* new timestamp is the highest priority in queue = head of queue */
    temp_tst = min_timestamp(*perm, *queue);
  }
  else  {
    /* find min(*perm, *queue) */
    temp_tst = min_timestamp(*perm, *queue);

    in_tst->log_time = log_time;
    in_tst->node     = node;
    in_tst->tsptr    = NULL;
```

```
    /* find min(temp_tst, new request);  (new request = in_tst) */
    temp_tst = min_timestamp(temp_tst, in_tst);
  }
  if (temp_tst->log_time != log_time)  {
    /*  new request has lower priority than perm and request at
     * head of queue */
   strcpy(answer, "/WAIT\n");
  }
  else  {
    if (temp_tst->node != node)  {
      /*  new request has lower priority than perm and request at
       * head of queue */
     strcpy(answer, "/WAIT\n");
    }
    else  {
      /* new request has the highest priority */
     strcpy(answer, "/QUERYING\n");
    }
  }
  free(in_tst);
  return answer;
}

/*-----------------------------------------------------------------------
 * min_timestamp: return the minimum timestamp (comparing between tst1 nd
 *                tst2).
 * tst1 = 1st timestamp
 * tst2 = 2nd timestamp
 *-----------------------------------------------------------------------*/
struct timestamp* min_timestamp(struct timestamp *tst1, struct timestamp *tst2)  {
  if (tst1->log_time < tst2->log_time)  {
    return tst1;
  }
  else if (tst1->log_time == tst2->log_time)  {
   if (tst1->node < tst2->node)
      return tst1;
    else
      return tst2;
  }
  else  {
    return tst2;
  }
}

/*-----------------------------------------------------------------------
 * chk_next_request: After removing a request from PERM, check if there is
 *                any request left in QUEUE.  If there is a request in
 *                QUEUE, move the smallest request from QUEUE to PERM
 *                and return YES.  Otherwise, return NO.
 * perm  = PERM table kept the LLM locking this RLC.
 * queue = QUEUE table kept requests from LLM.
 *-----------------------------------------------------------------------*/
int chk_next_request(struct timestamp **perm, struct timestamp **queue)  {
 int answer;

  answer = NO;
  if (*queue != NULL)  {
     mv_to_tst(queue, perm , (*queue)->log_time, (*queue)->node);
```

```
    answer = YES;
  }
  return answer;
}

/*------------------------------------------------------------------------
 * query_server: send /QUERY message to the LLM kept in PERM to see if the
 *               LLM has already got a license.
 * tab      = table kept LLMs/USERs
 * node     = node number of LLM
 * log_time = logical time of the LLM
 * snd_msg  = total number of sending messages (messages sent to LLMs/Users).
 *------------------------------------------------------------------------*/
void query_server(struct table *tab, int node, ulong *log_time, int *snd_msg)  {
  struct table *run_tab;
  char         out_str[MAX_CHAR];
  char         time_str[MAX_LINE];

  run_tab = get_tab_from_node(tab, node);
  if (run_tab == NULL)  {  /* could not find the information of this node */
     printf("<tcpserv> query_server error at run_tab == NULL\n");
  }
  else  {
    strcpy(out_str, "/QUERY\n");
    itoa(*log_time, time_str);
    strcat(out_str, time_str);
    strcat(out_str, "\n");
    write(run_tab->new_fd, &out_str, sizeof(out_str));
    (*snd_msg)++;
  }
  return;
}
```

**tcpcli1.c**

```
#include "inet.h"
#include "struct.h"
#include "sem.h"      /* library containing semaphore sys. calls */

extern struct kcot* create_coterie(int, int, struct serv_table *, int *);
extern void print_serv_tab(struct serv_table *);
extern void print_kcot(struct kcot *);
extern void print_llist(struct llist *);
extern void print_cotsel_tab(struct cot_chosen *);
extern void print_active_u_s(struct active_user_server *);
extern int serv_connect(int);
extern void put_on_tab(struct table **, int, char *);
extern void put_on_serv_tab(struct serv_table **, int, char *, int);
extern void put_on_tst_asc(struct timestamp **, ulong, int);
extern void put_on_active_u_s(struct active_user_server **, int, int);
extern void del_from_tab(struct table **, int);
extern void del_from_cotsel(struct cot_chosen **, int, int);
extern void del_from_active_u_s(struct active_user_server **, int, int);
extern void upd_serv_tab_node(struct serv_table *);
extern void upd_tab_node(struct table *);
extern void mv_to_servlist(struct serv_table **, struct serv_table **, int, char *, int);
extern void mv_to_tst(struct timestamp **, struct timestamp **, ulong, int);
extern struct cot_chosen* get_data_from_cotsel_node(struct cot_chosen *, int);
extern struct timestamp* get_data_from_tst_node(struct timestamp *, int);
extern int get_serv_from_active_u_s(struct active_user_server *, int);
extern struct kcot* get_quorum_by_kcot_num(struct kcot *, int);
extern int chk_node_in_quorum(struct kcot *, int);
extern void free_tab(struct table **);
extern void free_tst_tab(struct timestamp **);
extern void free_cotsel_tab(struct cot_chosen **);
extern void free_kcot(struct kcot **);
extern void free_active_u_s(struct active_user_server **);
extern ulong adjust_log_time(ulong, ulong);
extern struct kcot* select_quorum(struct kcot*, struct timestamp *, struct timestamp *, struct
cot_chosen **);
extern struct serv_table* get_serv_from_node(struct serv_table*, int);
extern struct table* get_tab_from_node(struct table *, int);
extern int chk_node_from_tst(struct timestamp *, int);
extern void itoa(int, char *);
void initialize_var(void);
void free_memory(void);
struct serv_table* make_connect(char *, int *, int *, int *, int *, char **, int *);
void close_all_serv_connect(struct serv_table *, char *);
void close_all_user_connect(struct table *, char *, int *);
void contact_user(int, char *, struct serv_table *);
void contact_RLC(struct serv_table *, struct kcot **);
struct kcot *chk_license(struct serv_table *, struct kcot **, struct timestamp **, struct
timestamp **, struct cot_chosen **);
int chk_get_license(struct kcot *, struct timestamp *, struct cot_chosen *, struct kcot **);
void exit_rtn(struct table **, int, int *);
int req_service(int);
void service_user(struct timestamp **, struct table *, int, struct serv_table *, int *);
void release_server(struct cot_chosen *, struct kcot **, struct serv_table *, struct timestamp
**, struct timestamp **);
void release_tst(struct serv_table *, struct timestamp **, int);
```

```
void release_unneeded_lock(struct cot_chosen **, struct timestamp **, struct timestamp **,
struct serv_table *, struct kcot **);

/* global shared variables */
int *semid1, *semid2, *semid3, *semid4, *semid5, *semid6; /* Semaphore id */
int *request_no;    /* total number of users' requests */
int *get_license;   /* 0 = don't get licenses;                   */
                    /* otherwise = number of licenses got from RLCs */
int *terminate;     /* NO = not terminate, YES = terminater (end program */
ulong *log_time;    /* logical time of remote license controller which will initiate to 0 */
int *release_no;    /* 0 = don't release license servers; otherwise, release */
char **exchange_mesg;   /* messges that exchange between child processes */
int *active_users; /* the number of users who are using the s/w */
int *s_snd_msg; /* message # sending to RLC */
int *s_rec_msg; /* message # receiving from RLC */

main(int argc, char *argv)
{
  ulong   in_logtime;
  ulong   temp_logtime;

  /* coterie variables */
  struct serv_table   *cur_serv;  /* table kept servers' addresses */
  struct kcot         **cot;      /* coteries */
  int        N, k;      /* total number of hosts, and number of licenses */
  int        kcot_size; /* k-coterie size(total # of quorums in k-coteries) */
  int        max_fd;

  /* shared memory and semaphore variables */
  int shmid1, shmid2, shmid3, shmid4; /* Id of sharedm memory segments */
  char  *shmaddr1;            /* address of shared memory 1 */
  char  *shmaddr2;            /* address of shared memory 2 */
  char  *shmaddr3;            /* address of shared memory 3 */
  char  *shmaddr4;            /* address of shared memory 4 */
  int  pid;          /* Process id after fork call */
  int  status;       /* Exit status of child process */
  int  i, j;
  char str[80];       /* character string for root command */

  /* connection variables */
  int              own_port;
  char               *own_addr;
  int              sockfd, newsockfd, max_desc, clilen;
  struct sockaddr_in   cli_addr;
  fd_set             readset;
  int                      init_msg;  /* message # when initializing system */

  init_msg  = 0;
  N       = 0;
  k       = 0;
  kcot_size = 0;

  if (argc < 2)  {
     printf("Usage:  tcpcli <host_addr[version]> \n");
     exit(0);
  }

  initialize_var();
```

```
/* network connection => this local license manager try to connect
 * to all license servers */

own_addr = (char *) shmalloc(sizeof(char[ADDR_LEN]));
cur_serv = (struct serv_table *) shmalloc(sizeof(struct serv_table));
cur_serv = make_connect(argv[1], &N, &k, &max_fd, &own_port, &own_addr, init_msg);

/* update node number of cur_serv table */
upd_serv_tab_node(cur_serv);

/* k must not be greater than N */
if (k > N) {
  printf("<tcpcli> licenses(%d) > hosts(%d)\n", k, N);
  printf("<tcpcli> adjust licenses to be equal to hosts = %d\n", N);
  k = N;
}

/* create coterie */
cot = (struct kcot **) shmalloc(sizeof(struct kcot *));
*cot = NULL;
*cot = create_coterie(N, k, cur_serv, &kcot_size);
if (*cot == NULL) {
  /*
   * Parent cleans up the IPC entries. This is critically important;
   */
  close_all_serv_connect(cur_serv, own_addr);
  free_serv_tab(&cur_serv);
  shfree((char *) *cot);
  shfree((char *) cur_serv);
  shfree(own_addr);
  free_memory();
  printf("Number of messages occurred in the initialization step = %d\n", init_msg);
  printf("Number of Sending messages to RLC = %d\n", *s_snd_msg);
  printf("Number of receiving messages from RLC = %d\n", *s_rec_msg);
  return;
}

printf("SYSTEM IS COMPLETELY INITIALIZED !!!\n");
for (i = 1; i <= 2; i++) {
  if ((pid = fork()) == -1) {
    perror("Fork failed");
    exit(1);
  }
  if (pid == CHILD) {
    switch(i) {
     case 1: contact_user(own_port, own_addr, cur_serv);
          break;
     case 2: contact_RLC(cur_serv, cot);
          break;
    }
    exit(0);
  }
}

while (!(*terminate)) {
  strcpy(str, "");
  scanf("%s", &str);
```

```
    if (strcmp(str, "/terminate") == 0)  {
       (*terminate) = YES;
       /* Now wait for the child processes to finish */
       for (i = 1; i <= 2; i++) {
         pid = wait(&status);
       }

       close_all_serv_connect(cur_serv, own_addr);
       free_serv_tab(&cur_serv);
       free_kcot(cot);
       shfree((char *) *cot);
       shfree((char *) cur_serv);
       shfree(own_addr);
       free_memory();
       printf("Number of messages occurred in the initialization step = %d\n", init_msg);
       printf("Number of Sending messages to RLC = %d\n", *s_snd_msg);
       printf("Number of receiving messages from RLC = %d\n", *s_rec_msg);
    }  /* if terminate */
  }  /* while terminate */
  return;
}

/*--------------------------------------------------------------------
 * initialize_var: allocate memory for variables and semaphore variables,
 *                and initialize the values for those variables.
 *--------------------------------------------------------------------*/
void initialize_var(void)  {
  /*
   * Initialize the semaphore to 1 (binary semaphore)
   */
  semid1 = (int *) shmalloc(sizeof(int));
  semid2 = (int *) shmalloc(sizeof(int));
  semid3 = (int *) shmalloc(sizeof(int));
  semid4 = (int *) shmalloc(sizeof(int));
  semid5 = (int *) shmalloc(sizeof(int));
  semid6 = (int *) shmalloc(sizeof(int));

  *semid1 = sem_create((key_t)SEMKEY_VAL1, 1);
  if (*semid1 == -1) {
     printf("Semaphore(1) initialization failed.\n");
    exit(1);
  }

  *semid2 = sem_create((key_t)SEMKEY_VAL2, 1);
  if (*semid2 == -1) {
     printf("Semaphore(2) initialization failed.\n");
    exit(1);
  }

  *semid3 = sem_create((key_t)SEMKEY_VAL3, 1);
  if (*semid3 == -1) {
     printf("Semaphore(3) initialization failed.\n");
    exit(1);
  }

  *semid4 = sem_create((key_t)SEMKEY_VAL4, 1);
  if (*semid4 == -1) {
     printf("Semaphore(4) initialization failed.\n");
```

```
    exit(1);
  }

  *semid5 = sem_create((key_t)SEMKEY_VAL5, 1);
  if (*semid5 == -1) {
    printf("Semaphore(5) initialization failed.\n");
    exit(1);
  }

  *semid6 = sem_create((key_t)SEMKEY_VAL6, 1);
  if (*semid6 == -1) {
    printf("Semaphore(6) initialization failed.\n");
    exit(1);
  }

  request_no  = (int *) shmalloc(sizeof(int));
  get_license = (int *) shmalloc(sizeof(int));
  log_time    = (ulong *) shmalloc(sizeof(ulong));
  release_no  = (int *) shmalloc(sizeof(int));
  exchange_mesg  = (char **) shmalloc(sizeof(char *));
  *exchange_mesg = (char *) shmalloc(sizeof(char[MAX_CHAR]));
  active_users   = (int *) shmalloc(sizeof(int));
  terminate      = (int *) shmalloc(sizeof(int));
  s_snd_msg      = (int *) shmalloc(sizeof(int));
  s_rec_msg      = (int *) shmalloc(sizeof(int));

  *request_no  = 0;
  *get_license = 0;
  *log_time    = 0;
  *release_no  = 0;
  strcpy(*exchange_mesg, "");
  *active_users = 0;
  *terminate   = NO;
  *s_snd_msg   = 0;
  *s_rec_msg   = 0;
  return;
}

/*-------------------------------------------------------------------------
 * free_memory: free all allocated memory before terminating from the program.
 *-------------------------------------------------------------------------*/
void free_memory(void)  {
  shfree((char *) request_no);
  shfree((char *) get_license);
  shfree((char *) log_time);
  shfree((char *) terminate);
  shfree((char *) release_no);
  shfree(*exchange_mesg);
  shfree((char *) *exchange_mesg);
  shfree((char *) active_users);
  /*
   * Parent cleans up the IPC entries. This is critically important;
  */
  sem_rm(*semid1);
  sem_rm(*semid2);
  sem_rm(*semid3);
  sem_rm(*semid4);
  sem_rm(*semid5);
```

```
  sem_rm(*semid6);
  shfree((char *) semid1);
  shfree((char *) semid2);
  shfree((char *) semid3);
  shfree((char *) semid4);
  shfree((char *) semid5);
  shfree((char *) semid6);
  return;
}

/*-----------------------------------------------------------------------------
 * contact_user: communicate with users (send messages to or receive from
 *               users. Handle all jobs involving users.
 * own_port  ==> port number of this local license manager
 * own_addr  ==> address of this local license manager
 * cur_serv  ==> the table kept server information
 *----------------------------------------------------------------------------*/
void contact_user(int own_port, char *own_addr, struct serv_table *cur_serv)  {
  fd_set              readset;
  int                 sockfd, newsockfd, max_desc, clilen;
  struct timeval      wait_time;
  struct sockaddr_in  cli_addr;
  char                u_addr[ADDR_LEN];

  struct table         *user_tab;  /* table kept users' addresses */
  struct table         *tail;      /* temporary table */

  /* keep the timestamps of users requested for services */
  struct timestamp     *user_tst;
  struct timestamp     *run_tst;

  /* active_user_server table kept user nodes who got license and severs
   * that are used by these users.  This will be used in the case that
   * a user sends /exit without waiting for services */
  struct active_user_server  *active_nodes;

  int                 serv_num;

  char                mesg[MAX_CHAR];
  char                in_str[MAX_CHAR];
  char                in_str1[MAX_CHAR];
  char                in_str11[MAX_CHAR];
  char                in_str12[MAX_CHAR];
  char                in_str2[MAX_CHAR];
  char                out_str[MAX_CHAR];
  char                sockfd_str[MAX_LINE];

  int                 serv_node;
  char                serv_str[MAX_LINE];
  ulong               in_logtime;
  ulong               temp_logtime;
  int                 temp_fd;
  int                 answer;
  int                 j, n;
  int                 temp_get_license;
  char                time_str[MAX_CHAR];
  int                 u_snd_msg;
  int                 u_rec_msg;
```

```
int                     user_num;
int                     tot_req_no;

 u_snd_msg = 0;
 u_rec_msg = 0;
 user_num  = 0;
 tot_req_no = 0;

/* make a connection with users */
 sockfd = serv_connect(own_port + 1);

if (listen(sockfd, MAX_USER) < 0)
   err_sys("listen error");

user_tab = NULL;
user_tst = NULL;
active_nodes = NULL;

wait_time.tv_sec=5;
wait_time.tv_usec=0; /* Wait time is 5 seconds */

for ( ; ; ) {
  /*
   * Wait for a connection from a client process.
   */
   FD_ZERO(&readset);
   FD_SET(sockfd, &readset);

   if (user_tab == NULL)  {
      max_desc = sockfd;
   }
   else {
      for (tail = user_tab; tail != NULL; tail = tail->tptr) {
        FD_SET(tail->new_fd, &readset);
        if (max_desc < tail->new_fd) {
           max_desc = tail->new_fd;
        }
      }
   }

   /* see if any descriptors are ready to be read, check at most 5 sec. */
   if ((j = select(max_desc+1,&readset,NULL,NULL,&wait_time)) > 0) {

      sem_wait(*semid3);
      (*log_time) += D;   /* increment logical time */
      sem_signal(*semid3);

      /* check if a user asks for connection */
      if (FD_ISSET(sockfd, &readset)) {
         clilen    = sizeof(cli_addr);
          newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
         if (newsockfd > 0)  {
           strcpy(in_str, "");
           read(newsockfd, &in_str, MAX_CHAR);
           u_rec_msg++;
           user_num++;
           strcpy(mesg, in_str);
           strcpy(in_str1, strtok(in_str, " \n"));
```

```
      strcpy(in_str2, strtok(NULL, "\n"));
      if (strcmp(in_str1,"/addr") == 0)
        strcpy(u_addr, in_str2);
      else
        strcpy(u_addr, "");

      put_on_tab(&user_tab, newsockfd, u_addr);
    }
  }

  /* check to see if any client wants service */
  tail = user_tab;
  while (tail != NULL) {
    strcpy(in_str, "");
    if (FD_ISSET(tail->new_fd, &readset)) {
      read(tail->new_fd, &in_str, MAX_CHAR);
      u_rec_msg++;
      strcpy(mesg, in_str);
      strcpy(in_str1, strtok(in_str, " \n"));
      strcpy(in_str2, strtok(NULL, "\n"));
      in_logtime = atoi(in_str2);
      sem_wait(*semid3);
        *log_time = adjust_log_time(*log_time, in_logtime);
      sem_signal(*semid3);

      if (strcmp(in_str1,"/exit") == 0)  {
        /* release servers if this user already locked them */
        serv_node = get_serv_from_active_u_s(active_nodes, tail->node);
        if (serv_node != 0)  {  /* this user already got license */
          sem_wait(*semid6);
          (*active_users)--;
          sem_signal(*semid6);

          del_from_active_u_s(&active_nodes, tail->node, serv_node);

          itoa(serv_node, serv_str);
          if ((active_nodes == NULL) && (user_tst == NULL))  {
            strcpy(out_str, "/release_all-");
          }
          else  {
            strcpy(out_str, "/release_one-");
          }
          strcat(out_str, serv_str);
          strcat(out_str, "\n");

          sem_wait(*semid5);
          if (strcmp(*exchange_mesg, "") == 0)  {
              strcpy(*exchange_mesg, out_str);
          }
          else  {
              strcat(*exchange_mesg, out_str);
          }
          sem_signal(*semid5);

          sem_wait(*semid4);
          (*release_no)++;
          sem_signal(*semid4);
        }
```

```
/* remove requests of this users from user_tst and reduce
 * request_no */
answer = chk_node_from_tst(user_tst, tail->node);
if (answer == YES) {
  sem_wait(*semid1);
  (*request_no)--;
  if (*request_no < 0) {
    printf("<tcpcli> RLC error at request_no < 0 => request_no = %d\n",
    request_no);
    *request_no = 0;
  }
  sem_signal(*semid1);
  del_from_tst_by_node(&user_tst, tail->node);
}

temp_fd = tail->new_fd;
tail = tail->tptr;
exit_rtn(&user_tab, temp_fd, &u_snd_msg);
continue;
}
else if (strcmp(in_str1,"/finish") == 0) {

  serv_node = get_serv_from_active_u_s(active_nodes, tail->node);
  if (serv_node != 0) { /* this user already got a license */

    sem_wait(*semid6);
    (*active_users)--;
    sem_signal(*semid6);

    del_from_active_u_s(&active_nodes, tail->node, serv_node);
    itoa(serv_node, serv_str);
    if ((active_nodes == NULL) && (user_tst == NULL)) {
      strcpy(out_str, "/release_all-");
    }
    else {
      strcpy(out_str, "/release_one-");
    }
    strcat(out_str, serv_str);
    strcat(out_str, "\n");

    sem_wait(*semid5);
    if (strcmp(*exchange_mesg, "") == 0) {
      strcpy(*exchange_mesg, out_str);
    }
    else {
      strcat(*exchange_mesg, out_str);
    }
    sem_signal(*semid5);

    sem_wait(*semid4);
    (*release_no)++;
    sem_signal(*semid4);

    strcpy(out_str, "/ready\n");
    sem_wait(*semid3);
    itoa(*log_time, time_str);
    sem_signal(*semid3);
```

```
            strcat(out_str, time_str);
            strcat(out_str, "\n");
            write(tail->new_fd, &out_str, sizeof(out_str));
            u_snd_msg++;
          }
          else {
              printf("<tcpcli> USER error at get_serv_node_from_active_u_s \n");
          }
        }
        else if (strcmp(in_str1,"/addr") == 0) {
            strcpy(tail->addr, in_str2);
        }
        else if (strcmp(in_str1,"/run") == 0) {
          strcpy(out_str, "/wait\n");
          sem_wait(*semid3);
          itoa(*log_time, time_str);
          sem_signal(*semid3);
          strcat(out_str, time_str);
          strcat(out_str, "\n");
          write(tail->new_fd, &out_str, sizeof(out_str));
          u_snd_msg++;

          answer = chk_node_from_tst(user_tst, tail->node);
          if (answer == YES) {
            strcpy(out_str, "/duplicate\n");
            sem_wait(*semid3);
            itoa(*log_time, time_str);
            sem_signal(*semid3);
            strcat(out_str, time_str);
            strcat(out_str, "\n");
            write(tail->new_fd, &out_str, sizeof(out_str));
            u_snd_msg++;
          }
          else {
            sem_wait(*semid3);
            temp_logtime = *log_time;
            sem_signal(*semid3);
            put_on_tst_asc(&user_tst, temp_logtime, tail->node);
            sem_wait(*semid1);
            (*request_no)++;
            sem_signal(*semid1);
            tot_req_no++;
          }  /* else of if (answer == YES) */
        }
        else {
          strcpy(out_str, "<tcpcli> USER Unknown command: ");
          strcat(out_str, in_str1);
          strcat(out_str, "\n");
          write(tail->new_fd, &out_str, sizeof(out_str));
          u_snd_msg++;

        }
      } /* if FD_ISSET */
      tail = tail->tptr;
    } /* while */
  } /* if of select */

  sem_wait(*semid2);
```

```
temp_get_license = *get_license;
sem_signal(*semid2);
if (temp_get_license > 0)  {
  sem_wait(*semid5);
  if (strcmp(*exchange_mesg, "") != 0)  {
     strcpy(mesg, *exchange_mesg);
    strcpy(*exchange_mesg, "");
  }
  else  {
    strcpy(mesg, "");
  }
  sem_signal(*semid5);

  if (strcmp(mesg, "") != 0)  {
     strcpy(in_str, mesg);
    strcat(in_str, "END\n");

    while (strcmp(in_str, "END\n") != 0)  {
       strcpy(in_str1, strtok(in_str, " \n"));
       strcpy(in_str, strtok(NULL, ""));

       strcpy(in_str11, strtok(in_str1, "-\n"));
       strcpy(in_str12, strtok(NULL, "\n"));

       if (strcmp(in_str11, "/not_available") == 0)  {
          run_tst = user_tst;
          while (run_tst != NULL)  {
              tail = get_tab_from_node(user_tab, run_tst->node);
            if (tail != NULL)  {
               strcpy(out_str, "/not_available\n");
                sem_wait(*semid3);
                itoa(*log_time, time_str);
               sem_signal(*semid3);
               strcat(out_str, time_str);
               strcat(out_str, "\n");
                write(tail->new_fd, &out_str, sizeof(out_str));
               u_snd_msg++;
            }
            else  {
                printf("<tcpcli> USER error at /no_available\n");
            }
             run_tst = run_tst->tsptr;
          }  /* while run_tst */

          sem_wait(*semid2);
          (*get_license)--;
          sem_signal(*semid2);
       }
       else if (strcmp(in_str11, "/server") == 0)  {
          if (user_tst == NULL)  {
             /* get license but no more request so release all locks */

             sem_wait(*semid1);
             if ((*request_no) != 0)  {
                *request_no = 0;
             }
             sem_signal(*semid1);
```

```
                strcpy(out_str, "/release_one-");
                strcat(out_str, in_str12);
                strcat(out_str, "\n");
                sem_wait(*semid5);
                if (strcmp(*exchange_mesg, "") == 0)  {
                    strcpy(*exchange_mesg, out_str);
                }
                else  {
                    strcat(*exchange_mesg, out_str);
                }
                sem_signal(*semid5);

                sem_wait(*semid4);
                (*release_no)++;
                sem_signal(*semid4);
            }  /* user_tst == NULL */
            else  {
                serv_num = atoi(in_str12);
                service_user(&user_tst, user_tab, serv_num, cur_serv, &u_snd_msg);
                put_on_active_u_s(&active_nodes, user_tst->node, serv_num);
                sem_wait(*semid6);
                (*active_users)++;
                sem_signal(*semid6);
                del_from_tst(&user_tst, user_tst->log_time, user_tst->node);
            }

            sem_wait(*semid2);
            (*get_license)--;
            sem_signal(*semid2);
        }  /* else if (/server) */
        else  {
            strcpy(out_str, strtok(mesg, "END"));
            sem_wait(*semid5);
            if (strcmp(*exchange_mesg, "") == 0)  {
                strcpy(*exchange_mesg, out_str);
            }
            else  {
                strcat(out_str, *exchange_mesg);
                strcpy(*exchange_mesg, out_str);
            }
            sem_signal(*semid5);
            break;
        }
        strcpy(mesg, in_str);
      }  /* while strcmp */
    }  /* else mesg != 0 */
  }

  if ((*terminate) == YES)  {
    close_all_user_connect(user_tab, own_addr, &u_snd_msg);
    free_tab(&user_tab);
    free_tst_tab(&user_tst);
    free_active_u_s(&active_nodes);
    break;
  }
}  /* for */
```

```
    printf("\n\n************************************\n");
  printf("SYSTEM IS COMPLETELY TERMINATED\n\n");
    printf("************************************\n\n");
  printf("Number of Users = %d\n", user_num);
  printf("Number of requests from users = %d\n", tot_req_no);
  printf("Number of Sending messages to users = %d\n", u_snd_msg);
  printf("Number of receiving messages from users = %d\n", u_rec_msg);
  return;
}

/*-----------------------------------------------------------------------
 * contact_RLC: communicate with RLC, control messages between LLM and RLC, and
 *              handle all jobs involving RLC and LLM (especially, require and
 *              get a license)
 * cur_serv ==> the table kept server information
 * cot      ==> k-coteries
 *-----------------------------------------------------------------------*/
void contact_RLC(struct serv_table *cur_serv, struct kcot **cot) {
  static int available = 1;
  /* keep the set of RLCs which have agreeed by message OK. */
  struct timestamp  *yes;
  struct timestamp  *run_yes;

  /* keep the set of RLCs which have disagreeed by message WAIT */
  struct timestamp  *notnow;
  struct timestamp  *run_notnow;

  struct cot_chosen *cot_selected;
  struct cot_chosen *temp_cot;
  struct cot_chosen *run_cotsel;
  struct kcot       *permit_cot;  /* quorum selected for servicing users */
  struct llist      *run_list;
  int               permit_cotnum;
  char              mesg[MAX_CHAR];
  char              in_str[MAX_CHAR];
  char              in_str1[MAX_CHAR];
  char              in_str11[MAX_CHAR];
  char              in_str12[MAX_CHAR];
  char              in_str2[MAX_CHAR];
  char              out_str[MAX_CHAR];
  char              temp_str2[MAX_CHAR];
  int               server_node;
  char              serv_node_str[MAX_LINE];
  int               serv_port;
  char              serv_addr_str[MAX_LINE];
  char              port_str[MAX_LINE];
  int               j, n;
  int               temp_release_no;
  int               temp_request_no;
  int               temp_active_users;
  char              time_str[MAX_LINE];

  struct serv_table *run_serv;
  struct timeval    wait_time;
  fd_set            readset;
  int               max_desc;
  int               in_logtime;
  int               license_serv;
```

```
yes          = NULL;
notnow       = NULL;
cot_selected = NULL;
temp_cot     = NULL;
permit_cot   = NULL;

wait_time.tv_sec=5;
wait_time.tv_usec=0; /* Wait time is 5 seconds */

for ( ; ; ) {
  /*
   * Wait for a connection from a client process.
   */
  FD_ZERO(&readset);

  max_desc     = 0;
  if (cur_serv == NULL)  {
     printf("<tcpcli> CONTACT_RLC error at cur_serv == NULL\n");
     *terminate = YES;
     break;
  }
  else  {
     for (run_serv = cur_serv; run_serv != NULL; run_serv = run_serv->sptr)  {
       FD_SET(run_serv->new_fd, &readset);
       if (max_desc < run_serv->new_fd)  {
          max_desc = run_serv->new_fd;
       }
     }
  }

  /* see if any descriptors are ready to be read, check at most 5 sec. */
  if ((j = select(max_desc+1,&readset,NULL,NULL,&wait_time)) > 0) {
     sem_wait(*semid3);
     *log_time += D;
     sem_signal(*semid3);

    /* check to see if any server sends any message to this LLM */
    run_serv = cur_serv;
    while (run_serv != NULL)  {
      strcpy(in_str, "");
       if (FD_ISSET(run_serv->new_fd, &readset)) {
         read(run_serv->new_fd, &in_str, MAX_CHAR);
         (*s_rec_msg)++;
         strcpy(mesg, in_str);
         strcpy(in_str1, strtok(in_str, " \n"));
         strcpy(in_str2, strtok(NULL, "\n"));

         if (strcmp(mesg, "") != 0)  {
            in_logtime = atoi(in_str2);
            sem_wait(*semid3);
             *log_time = adjust_log_time(*log_time, in_logtime);
            sem_signal(*semid3);
         }

         if (strcmp(in_str1,"/OK") == 0)  {
            /* a server sends OK message to LLM to allow this LLM
             * to lock it */
```

```
        license_serv = 0;
          run_notnow = get_data_from_tst_node(notnow, run_serv->node);
          if (run_notnow != NULL)  {  /* server sent OK is in notnow */
            mv_to_tst(&notnow, &yes, run_notnow->log_time, run_notnow->node);
            sem_wait(*semid1);
             temp_request_no = *request_no;
            sem_signal(*semid1);
            if (temp_request_no <= 0)  {
              release_unneeded_lock(&cot_selected, &yes, &notnow, cur_serv, cot);
            }
            available = 1;

          }  /* if run_notnow != NULL */
          else  {  /* not in notnow */
            strcpy(out_str, "/release_lock\n");
            sem_wait(*semid3);
            itoa(*log_time, time_str);
            sem_signal(*semid3);
            strcat(out_str, time_str);
            strcat(out_str, "\n");
            write(run_serv->new_fd, &out_str, sizeof(out_str));
            (*s_snd_msg)++;
          }
        }  /* if in_str1 = OK */
        else if (strcmp(in_str1,"/QUERY") == 0)  {
          /* check if a user has been using services from this license
           * server.  If No, and this license server is in YES, move
           * from YES to NOTNOW and send /RELINQUISH message back to
           * this server.  Otherwise, do nothing */
          run_cotsel = get_data_from_cotsel_node(cot_selected, run_serv->node);
          if (run_cotsel == NULL)  {
            /* this server hasn't been used by users now */
             run_yes = get_data_from_tst_node(yes, run_serv->node);
            if (run_yes != NULL)  {
              /* move this server from yes to notnow */
               mv_to_tst(&yes, &notnow, run_yes->log_time, run_yes->node);
              strcpy(mesg, "/relinquish\n");
              sem_wait(*semid3);
              itoa(*log_time, time_str);
              sem_signal(*semid3);
              strcat(mesg, time_str);
              strcat(mesg, "\n");
              write(run_serv->new_fd, &mesg, sizeof(mesg));
              (*s_snd_msg)++;
            }
          }
        }
      }  /* if (FD_ISSET) */
      run_serv = run_serv->sptr;
    }  /* while (run_serv) */
  } /* if (j = select.. */

  sem_wait(*semid4);
  temp_release_no = *release_no;
  sem_signal(*semid4);
  if (temp_release_no > 0)  {
    sem_wait(*semid5);
     strcpy(mesg, *exchange_mesg);
```

```
strcpy(*exchange_mesg, "");
sem_signal(*semid5);
strcpy(in_str, mesg);
strcat(in_str, "END\n");

while (strcmp(in_str, "END\n") != 0) {
  strcpy(in_str1, strtok(in_str, " \n"));
  strcpy(in_str, strtok(NULL, ""));


  strcpy(in_str11, strtok(in_str1, "-\n"));
  strcpy(in_str12, strtok(NULL, "\n"));
  if (strcmp(in_str11,"/release_one")==0) {
     server_node = atoi(in_str12);
      temp_cot = get_data_from_cotsel_node(cot_selected, server_node);
    if (temp_cot == NULL) {
       printf("<tcpcli> contact_RLC /release_one server = %d was already released\n",
       server_node);
    }
    else {
       release_server(temp_cot, cot, cur_serv, &yes, &notnow);
        del_from_cotsel(&cot_selected, temp_cot->kcot_num, temp_cot->serv_selected);
      available = 1;
    }
    sem_wait(*semid4);
    (*release_no)--;
    sem_signal(*semid4);
  }
  else if (strcmp(in_str11,"/release_all") == 0) {
     server_node = atoi(in_str12);

    sem_wait(*semid1);
     temp_request_no = *request_no;
    sem_signal(*semid1);

    sem_wait(*semid6);
    temp_active_users = *active_users;
    sem_signal(*semid6);

     if ((temp_request_no <= 0) && (temp_active_users == 0)) {
       /* no more request(s) and no active users */
       free_cotsel_tab(&cot_selected);
      cot_selected = NULL;
      release_server(cot_selected, cot, cur_serv, &yes, &notnow);
      sem_wait(*semid4);
       (*release_no) = 0;
       sem_signal(*semid4);
      available = 1;
    }
    else {  /* still have request(s) from users */
        temp_cot = get_data_from_cotsel_node(cot_selected, server_node);
      if (temp_cot == NULL) {
         printf("<tcpcli> contact_RLC /release_all server = %d was already released\n",
         server_node);
      }
      else {
         release_server(temp_cot, cot, cur_serv, &yes, &notnow);
          del_from_cotsel(&cot_selected,temp_cot->kcot_num,temp_cot->serv_selected);
```

```
            available = 1;
          }
          sem_wait(*semid4);
          (*release_no)--;
          sem_signal(*semid4);
        }
      }
      else {
        strcpy(out_str, strtok(mesg, "END"));
        sem_wait(*semid5);
        if (strcmp(*exchange_mesg, "") == 0) {
          strcpy(*exchange_mesg, out_str);
        }
        else {
          strcat(out_str, *exchange_mesg);
          strcpy(*exchange_mesg, out_str);
        }
        sem_signal(*semid5);

        break;
      }
      strcpy(mesg, in_str);
    } /* while mesg != END */
  }

  sem_wait(*semid1);
  temp_request_no = *request_no;
  sem_signal(*semid1);
  if ((temp_request_no > 0) && (available == 1)) {

    /* check if there is any quorum in cot_selected will be
     * a selected quorum.  Check only the quorum that hasn't
     * serviced users (with selected_serv = 0) but already been
     * chosen and checked with license servers */
    license_serv = 0;
    if (cot_selected != NULL) {
      run_cotsel = cot_selected;
      while (run_cotsel != NULL) {
        if (run_cotsel->serv_selected == 0) {
          permit_cot = get_quorum_by_kcot_num(*cot, run_cotsel->kcot_num);
          if (permit_cot == NULL) {
            printf("<tcpcli> RLC error at get_quorum_by_kcot_num\n");
          }
          else { /* found the quorum in cot */
            license_serv = chk_get_license(permit_cot, yes, cot_selected, cot);
            if (license_serv != 0) {
              /* there is a license server available */
              run_cotsel->serv_selected = license_serv;
              break;
            } /* if license_serv != 0 */
          }
        }
        run_cotsel = run_cotsel->next;
      }
    }

    if (license_serv == 0) {
      permit_cot = chk_license(cur_serv, cot, &yes, &notnow, &cot_selected);
```

```
    if (permit_cot == NULL)  {
      /*** couldn't get a license; tell users to ask for services later ** */
      sem_wait(*semid5);
      if (strcmp(*exchange_mesg, "/not_available\n") != 0)  {
        if (strcmp(*exchange_mesg, "") == 0)  {
          strcpy(*exchange_mesg, "/not_available\n");
        }
        else  {
          strcat(*exchange_mesg, "/not_available\n");
        }

        sem_wait(*semid2);
        (*get_license)++;
        sem_signal(*semid2);
      }
      sem_signal(*semid5);
      available = 0;
      continue;
    }
    else  {
      /* there is a license available to service a user */
      temp_cot = cot_selected;
      while (temp_cot != NULL)  {
        if (temp_cot->kcot_num == permit_cot->kcot_num)  {
          license_serv = temp_cot->serv_selected;
          break;
        }
        temp_cot = temp_cot->next;
      }
    }
    if (temp_cot == NULL)  {
      printf("<tcpcli> RLC error at checking kcot_num of cot_selected and
      permit_cot\n");
    }
  }

  if (license_serv != 0)  {
    strcpy(mesg, "/server-");
    itoa(license_serv, serv_node_str);
    strcat(mesg, serv_node_str);
    strcat(mesg, "\n");

    sem_wait(*semid5);
    if ((strcmp(*exchange_mesg, "") == 0) ||
        (strcmp(*exchange_mesg, "/not_available\n") == 0))  {
      if (strcmp(*exchange_mesg, "/not_available\n") == 0)  {
        sem_wait(*semid2);
        (*get_license)--;
        sem_signal(*semid2);
      }
      strcpy(*exchange_mesg, mesg);
    }
    else  {
      strcat(*exchange_mesg, mesg);
    }
    sem_signal(*semid5);

    sem_wait(*semid1);
```

```
        (*request_no)--;
        if (*request_no < 0) {
            *request_no = 0;
        }
        temp_request_no = *request_no;
        sem_signal(*semid1);

        /* no more request, release unneeded lock at servers */
        if (temp_request_no == 0) {
            release_unneeded_lock(&cot_selected, &yes, &notnow, cur_serv, cot);
        }

        sem_wait(*semid2);
        (*get_license)++;
        sem_signal(*semid2);
      }
    }

    if ((*terminate) == YES) {
      free_cotsel_tab(&cot_selected);
      free_tst_tab(&yes);
      free_tst_tab(&notnow);
      break;
    }
  } /* for */
  return;
}

/*-----------------------------------------------------------------
 * make_connect: try to connect to all RLCS on the system.
 * filename = name of file kept the number of license and the addresses of
 *            all RLCs on the system.
 * N        = the total number of RLCs on the system.
 * k        = the number of licenses.
 * max_fd   = the maxinum value of file descriptor
 * own_port = port number of this LLM.
 * own_addr = the address of this LLM.
 * init_msg = the total messages occurred in the initialization process.
 *-----------------------------------------------------------------*/
struct serv_table* make_connect(char *filename, int *N, int *k, int *max_fd, int *own_port,
char **own_addr, int *init_msg) {
  FILE  *fp;              /* file pointer pointed opened file */
  char  str[MAX_LINE];      /* temporary array of characters */
  char  mesg[MAX_CHAR];     /* message really sent */
  char  tcp_addr[ADDR_LEN];
  char  str_port[PORT_LEN];
  int   tcp_port;
  int   sockfd;
  fd_set readset;
  struct sockaddr_in    serv_addr;

  struct serv_table   *serv_tab;
  struct serv_table   *cur_serv;
  struct serv_table   *tail;

  int   flag;
  int     j;
```

```
cur_serv = NULL;
serv_tab = NULL;
tail     = NULL;

flag = YES;

/* open host_addr file (as read-only) for addresses of file servers */
if ((fp = fopen(filename, "r")) == NULL) {
  printf("Cannot open file <%s>\n", filename);
  exit(1);
}

if (fgets(str, MAX_LINE, fp) != NULL) {
  /* the first line in the file is the number of licenses */
  *k = atoi(str);
}

/* read address and port number from file. Local license managers
 * will connect to remote license controllers whose addresses are in the
 * file */
while (fgets(str, MAX_LINE, fp) != NULL) {
  strcpy(tcp_addr, strtok(str, " \n"));
  strcpy(str_port, strtok(NULL, " \n"));
  tcp_port = atoi(str_port);
  if (flag == YES) {
    flag = NO;
    strcpy(*own_addr, tcp_addr);
    *own_port = tcp_port;
  }

  /* Fill in the structure "serv_addr" with the address of the
   * server that we want to connect with. */
  memset((char *) &serv_addr, 0, sizeof(serv_addr));
  serv_addr.sin_family      = AF_INET;
  serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
  serv_addr.sin_port        = htons(tcp_port);

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

  put_on_serv_tab(&serv_tab, sockfd, tcp_addr, tcp_port);

  /* Connect to the server. */
  if (connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
    close(sockfd);  /* close sockfd when connection fails */
    (*init_msg)++;
  }
  else {
    /* after connecting sucessfully, remove address of host from serv_tab
     * to cur_serv */
    mv_to_servlist(&cur_serv, &serv_tab, sockfd, tcp_addr, tcp_port);

    (*N)++;  /* add 1 to N (number of hosts ) */
    /* send host name of the local license controller (itself) to the
     * remote license controller which it is being connected to.
```

```
        * This name is read from file */
       strcpy(mesg, "/addr\n");
       strcat(mesg, *own_addr);
       strcat(mesg, "\n");
       write(sockfd, &mesg, sizeof(mesg));
       (*init_msg)++;
    }
  }
  fclose(fp);

  /* keep connecting to all license servers */
  while (serv_tab != NULL)  {
    for (tail = serv_tab; tail != NULL; tail = tail->sptr)  {
      /* Fill in the structure "serv_addr" with the address of the
       * server that we want to connect with. */
       memset((char *) &serv_addr, 0, sizeof(serv_addr));
      serv_addr.sin_family      = AF_INET;
      serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
      serv_addr.sin_port        = htons(tail->port);

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

      /* Connect to the server. */
       if (connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        printf("<tcpcli> can't connect to server with fd = %d, addr: %s, port: %d\n", sockfd,
       tail->addr, tail->port);
        close(sockfd);   /* close sockfd when connection fails */
        (*init_msg)++;
        sleep(5);
      }
      else  {
         /* after connecting sucessfully, remove address of host from
          * serv_tab to cur_serv */
         tail->new_fd = sockfd;
          *max_fd = sockfd;
         mv_to_servlist(&cur_serv, &serv_tab, tail->new_fd, tail->addr, tail->port);
          (*N)++;   /* add 1 to N (number of hosts ) */

         /* send host name of the local license controller (itself) to the
          * remote license controller which it is being connected to.
          * This name is read from file */
         strcpy(mesg, "/addr\n");
          strcat(mesg, *own_addr);
         strcat(mesg, "\n");
          write(sockfd, &mesg, sizeof(mesg));
          (*init_msg)++;
          break;       /* start checking from head of list (table) */
      }
    }  /* for */
  }  /* while */

  return(cur_serv);
}
```

```
/*---------------------------------------------------------------------------
 * close_all_serv_connect: close all connection to servers (RLCs on the system).
 * cur_serv = the table kept all license servers of the system.
 * addr     = address of the license server.
 *---------------------------------------------------------------------------*/
void close_all_serv_connect(struct serv_table *cur_serv, char *addr) {
  struct serv_table *run_serv;
  char out_str[MAX_CHAR];
  char temp_addr[MAX_CHAR];

  run_serv = cur_serv;

  while (run_serv != NULL) {
    strcpy(out_str, "(server) Disconnect from Local License Manager = ");
    strcat(out_str, addr);
    strcat(out_str, "\n");
    write(run_serv->new_fd, &out_str, sizeof(out_str));
    (*s_snd_msg)++;
    strcpy(out_str, "/terminate\n");
    write(run_serv->new_fd, &out_str, sizeof(out_str));
    (*s_snd_msg)++;
    close(run_serv->new_fd);
    run_serv = run_serv->sptr;
  }
  return;
}

/*---------------------------------------------------------------------
 * close_all_user_connect: close all connection with users
 * user_tab = the table kept information of users connected to this LLM.
 * u_snd_msg = total number of messages sent to users.
 *---------------------------------------------------------------------*/
void close_all_user_connect(struct table *user_tab, char *addr, int *u_snd_msg) {
  struct table *run_tab;
  char out_str[MAX_CHAR];
  char temp_addr[MAX_CHAR];

  run_tab  = user_tab;
  while (run_tab != NULL) {
    strcpy(out_str, "(user) Disconnect from Local License Manager = ");
    strcat(out_str, addr);
    strcat(out_str, "\n");
    write(run_tab->new_fd, &out_str, sizeof(out_str));
    (*u_snd_msg)++;
    strcpy(out_str, "/terminate\n");
    write(run_tab->new_fd, &out_str, sizeof(out_str));
    (*u_snd_msg)++;
    close(run_tab->new_fd);
    run_tab  = run_tab->tptr;
  }
  return;
}
```

```
/*-------------------------------------------------------------------
 * chk_license: check if there is a license available.
 * cur_serv  = the table kept the addresses of servers (RLCs)
 * cot       = k-coterie formed in initialization step.
 * yes       = table kept the RLCs sending OK (permission) to this LLM.
 * notnow    = table kept the RLCs sending WAIT to this LLM.
 * cot_selected = table kept quorums already selected or visited.
 *------------------------------------------------------------------*/
struct kcot* chk_license(struct serv_table *cur_serv, struct kcot **cot, struct timestamp *yes,
struct timestamp **notnow, struct cot_chosen **cot_selected)  {

  struct kcot       *cur_cot;
  struct cot_chosen *visited_cot;
  struct cot_chosen *run_cotsel;
  struct llist      *l_tail, *l_run;
  struct serv_table *serv_info;
  struct timestamp  *run_yes;

  int   pid;       /* Process id after fork call */
  int   reply;     /* answer got from remote license controller */
  int   license_serv;
  int   get_permit; /* 1 = get permission, NO = don't get permission */
  int   status;
  int   answer;
  int   temp_logtime;

  cur_cot      = NULL;
  visited_cot  = NULL;
  run_cotsel   = NULL;
  l_tail       = NULL;
  l_run        = NULL;
  serv_info    = NULL;
  get_permit   = NO;
  license_serv = 0;

  /* keep the quorums that are selected by other users in visited_cot so we
   * don't have to check it again */
  if (*cot_selected != NULL)  {
    run_cotsel = *cot_selected;
    while (run_cotsel != NULL)  {
      put_on_cotsel_asc(&visited_cot, run_cotsel->kcot_num, run_cotsel->serv_selected);
      run_cotsel = run_cotsel->next;
    }
  }

  while (!get_permit)  {
    cur_cot = select_quorum(*cot, *yes, *notnow, &visited_cot);
    if (cur_cot == NULL)  {
      break;
    }
    for (l_tail = cur_cot->q; l_tail != NULL; l_tail = l_tail->ptr)  {
      serv_info = get_serv_from_node(cur_serv, l_tail->node);
      if (serv_info == NULL)  {
        printf("<tcpcli> Error in chk_license get_serv_from_node!!!\n");
        continue;
      }
      reply = req_service(serv_info->new_fd);
```

```
      sem_wait(*semid3);
      temp_logtime = *log_time;
      sem_signal(*semid3);

      if (reply == WAIT) {
         put_on_tst_asc(notnow, temp_logtime, serv_info->node);
      }
      else if (reply == OK) {
         put_on_tst_asc(yes, temp_logtime, serv_info->node);
      }
      /* in the case that reply == QUERY => do nothing */
    } /* for (l_tail) */

    license_serv = chk_get_license(cur_cot, *yes, *cot_selected, cot);
    if (license_serv != 0) {
      get_permit = YES;
    }
    put_on_cotsel_asc(cot_selected, cur_cot->kcot_num, license_serv);
  } /* while (!get_permit) */

  free_cotsel_tab(&visited_cot);
  return cur_cot;
}

/*-----------------------------------------------------------------
 * req_service: request a service from RLCs. That is, sending a message
 *             /lock to an RLC in the selected quorum and wait for the
 *             answer. If RLC is available, this LLM can lock it (got
 *             /OK message). If getting /WAIT = the RLC has been locked.
 *             If getting "/QUERYING", asking if this LLM has got a license.
 * sockfd = the file descriptor of the RLC.
 *-----------------------------------------------------------------*/
int req_service(int sockfd) {
  char  logtime_str[MAX_LINE];
  int   in_logtime;
  char  mesg[MAX_CHAR];
  char  mesg1[MAX_CHAR];
  char  mesg2[MAX_CHAR];
  char  temp_mesg[MAX_CHAR];
  int   answer;

  strcpy(mesg, "/lock\n");
  sem_wait(*semid3);
  itoa(*log_time, logtime_str);
  sem_signal(*semid3);
  strcat(mesg, logtime_str);
  strcat(mesg, "\n");
  write(sockfd, &mesg, sizeof(mesg));
  (*s_snd_msg)++;
  read(sockfd, &mesg, MAX_CHAR);
  (*s_rec_msg)++;
  strcpy(temp_mesg, mesg);

  strcpy(mesg1, strtok(mesg, " \n"));
  strcpy(mesg2, strtok(NULL, "\n"));
  if (strcmp(mesg1, "/OK") == 0) {
    answer = OK;
  }
```

```
  else if (strcmp(mesg1, "/WAIT") == 0) {
    answer = WAIT;
  }
  else if (strcmp(mesg1, "/QUERYING") == 0) {
    answer = QUERY;
  }
  else {
    printf("get incorrect message from server (REQ_SERVICE)!!! mesg = %s\n", temp_mesg);
    answer = WAIT;  /* easiest default value */
  }
  sem_wait(*semid3);
  *log_time += D;
  in_logtime = atoi(mesg2);
  *log_time = adjust_log_time(*log_time, in_logtime);
  sem_signal(*semid3);
  return answer;
}

/*----------------------------------------------------------------------
 * exit_rtn: delete the user from table (tab) and close the connection with
 *           that user.
 * tab = table kept the information of all users connecting to this LLM.
 * fd  = file descriptor of the user asking for disconnection.
 * u_snd_msg = total number of messages sending to users.
 *----------------------------------------------------------------------*/
void exit_rtn(struct table **tab, int fd, int *u_snd_msg)  {
  char out_str[MAX_CHAR];

  del_from_tab(tab, fd);
  strcpy(out_str, "exit: ok\n");
  write(fd, &out_str, sizeof(out_str));
  (*u_snd_msg)++;
  close(fd);
  return;
}

/*----------------------------------------------------------------------
 * chk_get_license: check if the LLM can get a license by checking if all
 *                  RLCs in the selected quorum are contained in YES.
 * in_cot = the quorum has been selected.
 * in_yes = YES table kept all RLCs sending the OK messages to this LLM.
 * cot_selected = quorums has been selected or visited.
 * cot    = the k-coterie formed in the initialization step.
 *----------------------------------------------------------------------*/
int chk_get_license(struct kcot *in_cot, struct timestamp *in_yes, struct cot_chosen
*cot_selected, struct kcot **cot)  {
  struct llist *run_list, *temp_list;
  struct timestamp *run_yes;
  struct cot_chosen *run_cotsel;
  struct kcot *temp_kcot;
  int   found;
  int   license_serv;
  int   first;
  int   i;
  int   order_num;

  first = YES;
  license_serv = 0;
```

```
for (run_list = in_cot->q; run_list != NULL; run_list = run_list->ptr)  {
    run_yes = get_data_from_tst_node(in_yes, run_list->node);
   if (run_yes == NULL)  {
      found = NO;
      break;
   }
   else  {
      found = YES;
   }
}

if (found == YES)  {
   /* check to see that a server in this quorum must not contain in
    * cot_selected (compare with serv_selected).  Eg, {1}, {2} have
    * serviced users.  If {1,3} is selected, it cannot be used */
   for (run_list = in_cot->q; run_list != NULL; run_list = run_list->ptr)  {
     if (first == YES)  {
        license_serv = run_list->node;
        first = NO;
     }
     run_cotsel = cot_selected;
     while (run_cotsel != NULL)  {
       if (run_cotsel->serv_selected != 0)  {
           temp_kcot = get_quorum_by_kcot_num(*cot, run_cotsel->kcot_num);
         if (temp_kcot == NULL)  {
             printf("<tcpcli> chk_get_license error at get_quorum_by_kcot_num\n");
           license_serv = 0;
           break;
         }
         else  {
            for (temp_list = temp_kcot->q; temp_list != NULL; temp_list = temp_list->ptr)
              if (temp_list->node == run_list->node)  {
               license_serv = 0;
               break;               }
           }  /* for */
         }  /* else */
       }  /* if */

       if (license_serv == 0)  {
         break;
       }
        run_cotsel = run_cotsel->next;
     }  /* while */

     if (license_serv == 0)  {
       break;
     }
   }  /* for */
}
else {
   license_serv = 0;
}

/* random a license server from the selected quorum */
if (license_serv != 0)  {
   i = 0;
   order_num = rand() % in_cot->size;
```

```
    for (temp_list = in_cot->q; temp_list != NULL; temp_list = temp_list->ptr) {
      if (i == order_num) {
        license_serv = temp_list->node;
        break;
      }
      else {
        i++;
      }
    }
  }
  return license_serv;
}

/*------------------------------------------------------------------------
 * service_user: send the address of RLC to user then user can use the
 *               software on the license server (RLC).
 * user_tst = table of timestamps sent from users.
 * user_tab = table kept addresses of users.
 * node     = node number of the server (RLC) to be connected by a user.
 * cur_serv = table kept the addresses of all servers (RLC) on the system.
 * u_snd_msg = total number of messages sending to users.
 *------------------------------------------------------------------------*/
void service_user(struct timestamp **user_tst, struct table *user_tab, int node, struct erv_table
*cur_serv, int *u_snd_msg) {
  struct serv_table *serv_tab;
  struct table      *tab;
  char        out_str[MAX_CHAR];
  char        port_str[MAX_LINE];
  int         temp_node;
  char        time_str[MAX_LINE];
  temp_node = (*user_tst)->node;
  tab       = get_tab_from_node(user_tab, temp_node);
  if (tab == NULL) {
    printf("Service_User error!  Couldn't find the user = %d information \n", temp_node);
    return;
  }
  serv_tab = get_serv_from_node(cur_serv, node);
  if (serv_tab == NULL) {
    printf("Service_User error!  Couldn't find the server = %d information \n", node);
    return;
  }
  strcpy(out_str, "/success\n");
  strcat(out_str, serv_tab->addr);
  strcat(out_str, "-");
  itoa(serv_tab->port, port_str);
  strcat(out_str, port_str);
  strcat(out_str, "-");
  sem_wait(*semid3);
  itoa(*log_time, time_str);
  sem_signal(*semid3);
  strcat(out_str, time_str);
  strcat(out_str, "\n");

  write(tab->new_fd, &out_str, sizeof(out_str));
  (*u_snd_msg)++;
  return;
}
```

```
/*------------------------------------------------------------------
 * release_server: send /release message to all servers in yes and notnow table.
 * the parameters are:
 * in_cot = a quourm that was already chosen
 * cot      = the k-coteries
 * cur_serv = the table kept license servers' information
 * yes      = yes table which will keep the information of license server that
 *            allowed a user to lock it.
 * notnow   = notnow table which cannot allow a user to lock it
 *------------------------------------------------------------------*/
void release_server(struct cot_chosen *in_cotsel, struct kcot **cot, struct serv_table cur_serv,
struct timestamp **yes, struct timestamp **notnow) {
  struct timestamp  *run_yes;
  struct timestamp  *run_notnow;
  struct serv_table *run_serv;
  struct kcot       *temp_cot;
  struct llist      *run_list;
  char              out_str[MAX_CHAR];
  char              node_str[MAX_LINE];
  char              time_str[MAX_LINE];

  if (in_cotsel == NULL)  {  /* release_all */
    /* send /release to all server nodes in YES U NOTNOW */

    /* send /release_wait to all nodes in NOTNOW */
    release_tst(cur_serv, notnow, 2);

    /* send /release_lock to all nodes in YES */
    release_tst(cur_serv, yes, 1);
  }
  else  {
    /* send message /release_lock to all server nodes in in_cotsel (input
     * quorum). These server nodes are in YES */
     temp_cot = get_quorum_by_kcot_num(*cot, in_cotsel->kcot_num);
    if (temp_cot == NULL)  {
      printf("<tcpcli> RELEASE_SERVER error at get_quorum_by_kcot_num\n");
    }
    else  {
      run_list = temp_cot->q;
      while (run_list != NULL)  {
         run_yes = get_data_from_tst_node(*yes, run_list->node);

        if (run_yes == NULL)  {
          printf("<tcpcli> RELEASE_SERVER error at run_yes == NULL\n");
          run_list = run_list->ptr;
          continue;
        }

         run_serv = get_serv_from_node(cur_serv, run_yes->node);

        if (run_serv == NULL)  {
          printf("<tcpcli> RELEASE_SERVER error at run_serv == NULL\n");
          run_list = run_list->ptr;
          continue;

        }
```

```
        strcpy(out_str, "/release_lock\n");
        sem_wait(*semid3);
        itoa(*log_time, time_str);
        sem_signal(*semid3);
        strcat(out_str, time_str);
        strcat(out_str, "\n");

        write(run_serv->new_fd, &out_str, sizeof(out_str));
        (*s_snd_msg)++;
        del_from_tst_by_node(yes, run_yes->node);

        run_list = run_list->ptr;
      }
    }
  } /* else of if (in_cotsel == NULL) */
  return;
}

/*-------------------------------------------------------------------
 * release_tst: release servers in in_tst table
 * cur_serv = list kept the information of servers
 * in_tst   = timestamp table that keeps info. of server < YES or NOTNOW >
 * flag     = if in_tst = YES, flag = 1 (/release_lock)
 *            else   (in_tst = NOTNOW), flag = 2 (/release_wait)
 *-------------------------------------------------------------------*/
void release_tst(struct serv_table *cur_serv, struct timestamp **in_tst, int flag)  {
  struct timestamp  *run_tst;
  struct serv_table *run_serv;
  char              out_str[MAX_CHAR];
  char              time_str[MAX_LINE];

  /* send /release_... to all nodes in in_tst */
  run_tst = *in_tst;
  while (run_tst != NULL)  {
    run_serv = get_serv_from_node(cur_serv, run_tst->node);

    if (run_serv == NULL)  {
      printf("<tcpcli> RELEASE_TST error at run_serv == NULL\n");
      run_tst = run_tst->tsptr;
      continue;
    }

    if (flag == 1)  {
      strcpy(out_str, "/release_lock\n");
    }
    else if (flag == 2)  {
      strcpy(out_str, "/release_wait\n");
    }
    else  {
      printf("<tcpcli> RELEASE_TST error at flag != 1, 2\n");
      return;
    }

    sem_wait(*semid3);
    itoa(*log_time, time_str);
    sem_signal(*semid3);
    strcat(out_str, time_str);
    strcat(out_str, "\n");
```

```
    write(run_serv->new_fd, &out_str, sizeof(out_str));
    (*s_snd_msg)++;

    run_tst = run_tst->tsptr;
  }
  free_tst_tab(in_tst);
  return;
}

/*-------------------------------------------------------------------
 * release_unneeded_lock: release all locks in notnow (/release_wait) and
 *                        releas lock of servers in YES which are not in a
 *                        quorum that is now active
 *------------------------------------------------------------------*/
void release_unneeded_lock(struct cot_chosen **cot_selected, struct timestamp **yes, struct
timestamp **notnow, struct serv_table *cur_serv, struct kcot **cot) {
  struct cot_chosen *run_cotsel;
  struct cot_chosen *temp_cotsel;
  struct kcot       *run_cot;
  struct kcot       *temp_cot;
  struct llist      *run_list;
  struct timestamp  *run_yes;
  struct serv_table *run_serv;
  int               temp_kcot_num;
  int               temp_node;
  int               flag;
  char              out_str[MAX_CHAR];
  char              time_str[MAX_LINE];

  /* release all server nodes in notnow */
  release_tst(cur_serv, notnow, 2);

  /* check if nodes in the quorum that has been chosen but unqualified
   * (serv_selected = 0) are in YES.  If yes and these nodes are not
   * in an active quorum, release this server and remove it from YES.
   */
  run_cotsel = *cot_selected;
  while (run_cotsel != NULL)  {
    if (run_cotsel->serv_selected == 0)  {  /* unneeded quorum */
       run_cot = get_quorum_by_kcot_num(*cot, run_cotsel->kcot_num);
      if (run_cot == NULL)  {
         printf("<tcpcli> release_unneeded_lock error1 at get_quorum_by_kcot_num\n");
      }
      else  {  /* found the quorum in cot */
        /* check if a node in this unneeded quorum is in YES.  If yes
         * check if it is also in active quorums. */
        flag = 0;
        for (run_list = run_cot->q; run_list != NULL; run_list = run_list->ptr)  {

           run_yes = get_data_from_tst_node(*yes, run_list->node);
          if (run_yes != NULL)  {
            /* check if this node is in active quorums.  If no, remove
             * it from yes and release license */
            flag = 0;
            temp_cotsel = *cot_selected;
            while (temp_cotsel != NULL)  {
              if (temp_cotsel->serv_selected != 0)  {
```

```
                temp_cot = get_quorum_by_kcot_num(*cot, temp_cotsel->kcot_num);
              if (temp_cot == NULL)  {
                 printf("<tcpcli> release_unneeded_lock error2 at
                      get_quorum_by_kcot_num\n");
              }
              else  {  /* found the quorum in cot */
                  flag = chk_node_in_quorum(temp_cot, run_yes->node);
                if (flag == 1)  {
                  break;
                }
              }
            }  /* if temp_cotsel->serv_selected != 0 */

            temp_cotsel = temp_cotsel->next;

          }  /* while temp_cotsel */

          if (flag == 0)  {  /* not found in active quorums */
            /* release this server and remove it from YES */
              run_serv = get_serv_from_node(cur_serv, run_yes->node);

            if (run_serv == NULL)  {
               printf("<tcpcli> release_unneeded_lock error at run_serv == NULL\n");
            }
            else  {
               strcpy(out_str, "/release_lock\n");
               sem_wait(*semid3);
               itoa(*log_time, time_str);
               sem_signal(*semid3);
               strcat(out_str, time_str);
               strcat(out_str, "\n");
               write(run_serv->new_fd, &out_str, sizeof(out_str));
               (*s_snd_msg)++;

                 del_from_tst_by_node(yes, run_yes->node);
            } /* run_serv = NULL */
          }  /* if flag = 0 */
        }  /* if run_yes */
      }  /* for */
    }  /*  else */

     /* delete unneeded quorum record on cot_selected */
     temp_kcot_num = run_cotsel->kcot_num;
    temp_node      = run_cotsel->serv_selected;
    run_cotsel     = run_cotsel->next;
     del_from_cotsel(cot_selected, temp_kcot_num, temp_node);
    continue;
  }  /* if (run_cotsel->serv_selected == 0) */
  run_cotsel = run_cotsel->next;
}  /* while */
return;

}
```

**user.c**

```
#include        "inet.h"
#include        "struct.h"
#include        "length.h"

extern int get_port_from_file(char *, char **);
extern void itoa(int, char *);
void connect_to_server(int, char *, char *, ulong *, int *, int *);

main(int argc, char *argv)
{
  ulong log_time;   /* logical time initiates to 0 */

  struct sockaddr_in    serv_addr;
  int                   in_logtime;
  char                  logtime_str[MAX_LINE];
  char                  mesg[MAX_CHAR];
  char                  in_str[MAX_CHAR];
  char                  in_str1[MAX_CHAR];
  char                  in_str2[MAX_CHAR];
  int                   own_port;
  char                  *own_addr;
  int                   sockfd, newsockfd, max_desc, clilen;
  struct sockaddr_in    cli_addr;
  struct timeval        wait_time;
  fd_set                readset;
  int                   j;
  int                   serv_port;
  char                  serv_addr_str[MAX_LINE];
  char                  port_str[MAX_LINE];
  int                   rec_msg;
  int                           snd_msg;
  int                           not_avail_no;

  rec_msg = 0;
  snd_msg = 0;
  not_avail_no = 0;

  log_time = 0;

  if (argc < 2)  {
    printf("Usage:  user  <host_addr[version]> \n");
    exit(0);
  }

  /* network connection => users try to connect to local license managers
   * (file servers) in order to get services */

  /* get port number from file */
  own_addr = (char *) malloc(sizeof(char[ADDR_LEN]));
  own_port = get_port_from_file(argv[1], &own_addr);

  /* local license manager port number (which users can connect to)
   *  = port number of remote license controller + 1 */
  own_port++;
```

```
/* Fill in the structure "serv_addr" with the address of the
 * server (local license manager) that we want to connect with.  */
memset((char *) &serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family      = AF_INET;
serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
serv_addr.sin_port        = htons(own_port);

/* Open a TCP socket (an Internet stream socket). */
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)  {
   err_sys("<user> can't open stream socket\n");
}

/* Connect to the server. */
if (connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
   printf("<user> can't connect to server with port: %d\n", own_port);
   close(sockfd);
   err_sys("");
}
else  {

   /* send host name of the local license controller (itself) to the
    * remote license controller which it is being connected to.
    * This name is read from file */
   strcpy(mesg, "/addr\n");
   strcat(mesg, own_addr);
   strcat(mesg, "\n");
   write(sockfd, &mesg, sizeof(mesg));
   snd_msg++;
}

wait_time.tv_sec=5;
wait_time.tv_usec=0;       /* Wait time is 5 seconds */

printf("READY TO CONTINUE !!!!!\n");
printf("~~~~~~~~~~~~~~~~~~~~~~~~\n");
for ( ; ; )  {
   FD_ZERO(&readset);
   FD_SET(sockfd, &readset);
   FD_SET(0, &readset);

   /* see if any descriptors are ready to be read, check at most 5 sec. */
   /* see if there are input from keyboard or from socket */
   if ((j = select(sockfd+1,&readset,NULL,NULL,&wait_time)) > 0) {

      log_time += D;   /* increment logical time */

      /* case reading from stdin */
      if (FD_ISSET(0, &readset)) {   /* 0 = stdin */
        strcpy(mesg, "");
        gets(mesg);
        if (strlen(mesg) == 0)  {
          continue;
        }
        strcat(mesg, "\n");
        itoa(log_time, logtime_str);
        strcat(mesg, logtime_str);
        strcat(mesg, "\n");
        write(sockfd, &mesg, sizeof(mesg));
```

```
      snd_msg++;
    }

    /* case reading from the socket descriptor */
    if (FD_ISSET(sockfd, &readset)) {
      read(sockfd, &mesg, MAX_CHAR);
      rec_msg++;
      strcpy(in_str, mesg);
      strcpy(in_str1, strtok(in_str, " \n"));
      strcpy(in_str2, strtok(NULL, "\n"));

      if (strcmp(in_str1, "/wait") == 0) {
        printf("Please wait!! If you don't want to wait, enter < /exit >\n");
        in_logtime = atoi(in_str2);
        log_time = adjust_log_time(log_time, in_logtime);
      }
      else if (strcmp(in_str1, "/success") == 0) {
        strcpy(serv_addr_str, strtok(in_str2, "- \n"));
        strcpy(port_str, strtok(NULL, "- "));
        strcpy(logtime_str, strtok(NULL, "\n"));

        in_logtime = atoi(logtime_str);
        log_time = adjust_log_time(log_time, in_logtime);

        serv_port = atoi(port_str);
        connect_to_server(serv_port, serv_addr_str, own_addr, &log_time, &snd_msg,
&rec_msg);
        strcpy(mesg, "/finish\n");
        itoa(log_time, logtime_str);
        strcat(mesg, logtime_str);
        strcat(mesg, "\n");
        write(sockfd, &mesg, sizeof(mesg));
        snd_msg++;
      }
      else if (strcmp(in_str1, "/ready") == 0) {
        in_logtime = atoi(in_str2);
        log_time = adjust_log_time(log_time, in_logtime);
        printf("Enter </exit> to exit or </run> to use software\n");

      }
      else if (strcmp(in_str1, "/duplicate") == 0) {
        in_logtime = atoi(in_str2);
        log_time = adjust_log_time(log_time, in_logtime);
        printf("You already asked for a service, please wait!! or enter </exit> to quit\n");
      }
      else if (strcmp(in_str1, "/not_available") == 0) {
        in_logtime = atoi(in_str2);
        log_time = adjust_log_time(log_time, in_logtime);
        printf("No license available now. Please wait !!! or enter </exit> to quit\n");
        not_avail_no++;
      }
      else {
        printf("<user.c> message: %s\n", mesg);
      }
    }
  }
```

```
    /* check if client wants to quit from the program */
    if ((strcmp(mesg, "exit: ok\n") == 0) ||
      (strcmp(mesg, "/terminate\n") == 0))
      break;
  }
  close(sockfd);
  free(own_addr);
  printf("Number of messages sending to RLC/LLM = %d\n", snd_msg);
  printf("Number of messages receiving from RLC/LLM = %d\n", rec_msg);
  printf("Number of Not_available messages receiving from LLM = %d\n", not_avail_no);
  return;
}

/*----------------------------------------------------------------------
 * connect_to_server: user tries to connect to RLC to use a software. After
 *                    that, use software, and disconnect from the RLC.
 * serv_port = server port (port of RLC to be connected)
 * serv_addr_str = address of server (RLC)
 * log_time     = logical time of the user.
 * snd_msg      = total number of messages sending to RLCs/LLMs
 * rec_msg      = total number of messages receiving from RLCs/LLMs.
 *---------------------------------------------------------------------*/
void connect_to_server(int serv_port, char *serv_addr_str, char *own_addr, ulong *log_time,
int *snd_msg, int *rec_msg) {
  int              sockfd;
  struct sockaddr_in   serv_addr;
  char               mesg[MAX_CHAR];
  char               in_str[MAX_CHAR];
  char               in_str1[MAX_CHAR];
  char               in_str2[MAX_CHAR];
  int              in_logtime;
  char               logtime_str[MAX_LINE];


  /* Fill in the structure "serv_addr" with the address of the
   * server that we want to connect with. */
  memset((char *) &serv_addr, 0, sizeof(serv_addr_str));
  serv_addr.sin_family      = AF_INET;
  serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
  serv_addr.sin_port        = htons(serv_port);

  /* Open a TCP socket (an Internet stream socket). */
  if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    printf("<user> can't open stream socket\n");
    return;
  }

  /* Connect to the server. */
  if (connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
    printf("<user> can't connect to server with fd = %d, addr: %s, port: %d\n", sockfd,
serv_addr_str, serv_port);
    close(sockfd);  /* close sockfd when connection fails */
  }
  else {
    strcpy(mesg, "/addr_user\n");
    strcat(mesg, own_addr);
    strcat(mesg, "\n");
    write(sockfd, &mesg, sizeof(mesg));
```

```
    (*snd_msg)++;
    printf("Enter </run_software> - use S/W or </exit> - disconnect server\n");
    printf("Enter: ");
    strcpy(mesg, "");
    gets(mesg);
    if ((strcmp(mesg, "/run_software") != 0)  && (strcmp(mesg, "/exit") != 0))  {
        printf("Wrong Command!!! Disconnect to server automatically\n");
        printf("Enter </run> to use software, or </exit> to quit from the program\n");

    }
    if (strcmp(mesg, "/run_software") == 0) {
       /* connecting to the license server sucessfully */
       /* system();  */
      strcat(mesg, "\n");
       itoa(*log_time, logtime_str);
      strcat(mesg, logtime_str);
      strcat(mesg, "\n");
      write(sockfd, &mesg, sizeof(mesg));
      (*snd_msg)++;

      *log_time += D;
      read(sockfd, &mesg, MAX_CHAR);
      (*rec_msg)++;
      printf("%s\n", mesg);

      *log_time += D;
      read(sockfd, &mesg, MAX_CHAR);
      (*rec_msg)++;
      strcpy(in_str, mesg);
      strcpy(in_str1, strtok(in_str, " \n"));
      strcpy(in_str2, strtok(NULL, "\n"));

      strcat(in_str1, "\n");
      printf("%s\n", in_str1);

       in_logtime = atoi(in_str2);
        *log_time = adjust_log_time(*log_time, in_logtime);
    }
    strcpy(mesg, "/exit\n");
    itoa(*log_time, logtime_str);
    strcat(mesg, logtime_str);
    strcat(mesg, "\n");
    write(sockfd, &mesg, sizeof(mesg));
    (*snd_msg)++;
    close(sockfd);
  }
  return;
}
```

# VITA

Sopana Prohmbhadra

Candidate for the Degree of

Master of Science

Thesis: **DISTRIBUTED LICENSE SERVER**

Major Field: Computer Science

Biographical:

Personal Data: Born in Bangkok, Thailand, March 27, 1969, a daughter of Pinai and Sudsanguan Prohmbhadra.

Education: Graduated from Wat Sungwet School, Bangkok, Thailand in April 1987 and received Bachelor of Science degree in Computer Science from Thammasat University, Bangkok, Thailand in February 1991. Completed the requirements for the Master of Science of Science degree with a major in Computer Science at Oklahoma State University in December 1995.

Experience: Employed as a programmer in information systems department by Toyota Motor Thailand Co., Ltd., Samut Prakan, Thailand from April 1991 to July 1992.