METHOD FOR PROTECTING AGAINST THE UNAUTHORIZED USE OF SOFTWARE IN A COMPUTER NETWORK ENVIRONMENT

BACKGROUND OF THE INVENTION

1. FIELD OF THE INVENTION

The present invention relates to a method for protecting against the unauthorized use of a software application in a computer network environment.

2. ART BACKGROUND

A computer network is typically an interconnection of machines or agents over links or cables. The open access characteristics of a computer network presents opportunities for the unauthorized copying of software, thus eroding the licensing revenue potential of software developers. Traditionally, either the entire network must be licensed (commonly referred to as a site license), or each node where the software is run must be licensed (commonly referred to as a node license). A node refers to a single machine, agent or system in a computer network. A license is an authorization given by a software developer to a customer to use a software application in a specific manner.

A site license lets all users at a designated location or network use the software application, regardless of their position on the network. This flat-fee approach is an overkill for a low usage software application. A node license not only ties a software application to a particular machine in a network, but also is not cost effective for the infrequent use of a software application. See, for example, U.S. Patent No. 4,688,169. Furthermore, if new users of licensed nodes wish to use the software application, they are often required to purchase additional licenses.

An alternative to a site license or a node license is the concept of a concurrent usage license. A concurrent usage license restricts the number of users allowed to use a software application at any given time, regardless of their location on the network. Just as renters check out available copies of a

movie video from a video rental store, users on a network check out a software application from an agent on a first-come-first-serve basis. Thus, a concurrent usage license charges a fee for the use of a software application proportional to its actual use.

Methods to license a software application for concurrent use in a network environment are currently offered by Highland Software, Inc. and Apollo Computer, Inc. See, M. Olson and P. Levine, "Concurrent Access Licensing", *Unix Review*, September 1988, Vol. 6, No. 9. In general, the license for a software application is stored in a database controlled by a license server. A license server is a program that not only stores the license, but also verifies the user's credentials before checking out the license to the authenticated user. To protect against the authorized use, these methods to license concurrent usage rely on secured communications such as public/private key encryption. Under public/private key encryption, each user of the system has two keys, one of which is generally known to the public, and the other which is private. The private transformation using the private key is related to the public one using the public key but the private key cannot be computationally determined from the public key. See Denning, D., *Cryptography and Data Security*, Addison-Wesley, 1982. The encryption key is hidden in the license server to encrypt the database of licenses. Well designed public/private key encryption schemes are difficult to crack, especially if the license server is located in a trusted environment. A trusted environment is one whose access is limited to users having the proper credentials. However, a license server is more likely to be located at a customer's site and hence in an hostile environment. It follows that the license server is vulnerable to sophisticated intruders. Once the private key is decrypted, all sensitive information on the license server such as licenses are compromised.

It is therefore an object of the present invention to provide a more secure method to protect against the unauthorized use of software in a concurrent use licensing environment.

SUMMARY OF THE INVENTION

The present invention provides to the software application the verification and license check out functions which are normally performed by a license server. The preferred embodiment of the present invention comprises a computer network including a plurality of agents running at least one license server and at least one software application. The license server controls a database of an agent containing the license information for the software application. The license information is contained in a license token, and is stored in the database controlled by the license server. The license token is a special bit pattern or packet which is encrypted by the software vendor of the application software. The software application communicates with the license server through a licensing library. The licensing library is a collection of library routines that the software application invokes to request or renew a license from the license server. Before a software application obtains a license, the license token must be decoded by a license access module. The license access module, which is linked with the software application and the licensing library is a program that decodes the license token from a vendor specific format to a licensing library format.

When an user wishes to run a software application, the licensing library invokes a call to request a license token from the license server. In contrast to the prior art where the license server either grants or denies the request after verifying the user's credentials, the license server in the preferred embodiment of the present invention finds the correct license token for the software application and transmits the license token to the licensing library. The license access module attached to the licensing library decodes the licensing token. Routines in the licensing library coupled to the software application verify the license information before checking out the license and updating the license token. The license access module encodes the updated license token before returning it to the license server.

Because the verification and check out function of a license token are performed by a software application, the software application rather than the license server becomes the point of attack by unauthorized users. Reverse engineering the license access module is less rewarding than attacking the license server because the license access module reveals the contents of a fraction of a database of licenses. By the time most attackers crack the license access module, the software vendors would most likely introduce newer versions of the software application and new license access modules for them. Thus the present invention provides a more secure method for protecting against the unauthorized use of a software application in a computer network environment without modifying the underlying computer network.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 illustrates a network environment employing the present invention.

Figure 2 describes the architecture of a network licensing scheme employing the preferred embodiment of the present invention.

Figure 3 describes the installation of a license token in the preferred embodiment of the present invention.

Figure 4a illustrates the use of a license token to request a license from a license server in the preferred embodiment of the present invention.

Figure 4b illustrates the use of a license token to renew a license from a license server in the preferred embodiment of the present invention.

Figure 4c illustrates the use of a license token to release a license from a license server in the preferred embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

The following detailed description is divided into several sections. The first of these sections describes a general network environment for accessing a database of licensed software programs. Subsequent sections discuss the details of a method for protecting against the unauthorized use of a software application.

I. General Network Environment

Referring to **Figure 1**, computer network environment comprises a plurality of data processing devices identified generally by numerals **10** through 10^n (illustrated as **10**, **10'** and 10^n). These data processing devices may include terminals, personal computers, workstations, minicomputer, mainframes and even supercomputers. For the purposes of this Specification, all data processing devices which are coupled to the present invention's network are collectively referred to as "agents". It should be understood that the agents may be manufactured by different vendors and may also use different operating systems such as MS-DOS, UNIX, OS/2, MAC OS and others. Particular examples of suitable agents include machines manufactured by Sun Microsystems, Inc., Mountain View, Calif. Each of the agents has an input device such as a keyboard **11**, **11'** and 11^n or a mouse **12**, **12'** and 12^n. As shown, agents **10** through 10^n (illustrated as **10**, **10'** and 10^n) are interconnected for data transfer to one another by a common cable **13**. It will be appreciated by one skilled in the art that the common cable **13** may comprise any shared media, such as coaxial cable, fiber optics, radio channel and the like. Furthermore, the network resulting from the interconnection of the cable **13** and agents **10** through 10^n (illustrated as **10**, **10'** and 10^n) may assume a variety of topologies, such as ring, star, bus, and may also include a collection of smaller networks linked by gateways or bridges.

Referring again to Figure 1 is a license service 14. The license service 14 is a resource shared by every agent connected to the network. In the preferred embodiment of the present invention, the license service 14 comprises license servers 15 through 15^m (illustrated as 15, 15' and 15^m) and databases 17 through 17^m (illustrated as 17, 17' and 17^m), where m is less than or equal to n. A license server is a program that runs on an agent with a memory storage capability. Each license server 15 (illustrated as 15, 15' and 15^m) communicates with a database 17 stored in memory on the agent over an interface 16 (illustrated as 16, 16' and 16^m). As will be described in detail below, the database 17 stores licensing information for various software applications which are purchased and authorized to run in the computer network environment. The license server is not limited to run on a specific agent, but can operate on any agent including the agent on which the user is to operate the application. Thus, any agent connected to the network may function as a license server as well as a device on which a user may operate application software. As will be described below, the license server does not perform verification of licenses of application software; rather the license server is passive and provides storing, locking, logging, and crash recovering function for the application software.

Figure 2 illustrates the architecture of a network licensing scheme of the present invention. The architecture comprises a database 18, database interface 19, license server 20, licensing library 24, License access module 27, license administration tool 21, license service binder 29, and license production tool 34.

The database 18 stores licensing information and application usage data. Preferably the database 18 comprises a plurality of records which contain the following information:

Database Element	Description
Unique Key Table	Keys for all other tables
Vendor Table	Vendor's ID and name
Product Table	Product number and name
Version Table	Version number and date
License Table	License #, exp date, total units
License Token Table	Stores encoded license token
Unit Group Table	A group's allocation of license
Group List Table	Name of the group
Allowed Users Table	Credentials of allowed users
Current License Use Table	Applications using a license
Lock Table	Locked records in database
Authorized administrator Table	Login names of administrators
License Operation Log Table	Administrator's log information
License Usage Log Table	Request handle plus Client Log
License Queue Log Table	License wait queue
Application Message Log Table	Application specific messages

A database interface 19 provides communication between the license server 20 and the database 18 in order to prevent concurrent access to the same database record by multiple users which can cause the data in the record to become corrupted. Thus, only the owner of the lock can read from and write to the locked record during the usage of the application.

The license server 20 operates on an agent and interfaces the database 18 to license administration tool 21, licensing library 24 and license service binder 29. The license server 20 communicates with the license administration tool 21, licensing library 24 and license service binder 29 via an interface 23. Preferably the interface 23 is a remote procedure call

mechanism which permits a process operating on one device or agent connected to the network to request a resource or service from a remote device or agent connected to the network. See A. Birrell and B. Nelson, "Implementing Remote Procedure Calls," *ACM Transaction on Computer Systems*, February 1984, Vol. 2, No. 1.

Multiple license servers may reside on multiple agents. Preferably the license server 20 operates in a background mode of the agent such that its operation is transparent to a user of that agent. More particularly, as will be described below, the license server 20 provides the following functions: 1) servicing the requests from the licensing library 24 for license token; (2) maintaining a wait queue for requests to the database 18 when no licensing units are available; (3) generating `locks` for exclusive access to database 18; and (4) providing access to information in the database 18.

The licensing library 24 is a set of library routines which enable the application 26 to request licensing service from the license server 20. Upon receiving the request for service from the licensing library 24, the license server 20 retrieves a license token from the database 18 and transmits it to the licensing library 24. The licensing library 24 is linked with the application 26 and communicates with the license server 20 over a path 28 with, preferably, a remote procedure call mechanism 23. Among the major library calls in the licensing library 24 is the application's request for a license from the license server 20. Other important library calls include the request to renew and to release a license. The use of the license token to accomplish the request for the various licensing service will be described in detail below.

The license access module (LAM) 27 is prepared by the software vendor 24 to decode the license token. Once decoded, the application 26 via routines in the licensing library verifies the licensing information in the license token and determines whether a license may be checked out. The LAM 27

also encodes the license token before the application returns it to the database 18 via license server 20. The license access module 27 is described in further detail below.

The license administration tool 21 is utilized by the network administrator to perform administrative functions relevant to the concurrent usage of a software application. The license administration tool 21 may run on any agent connected to the computer network. The license administration tool 21 is primarily used to install the license token into the database 18 through the license server 20. The functionality of the license administration tool 21 includes: (1) starting or terminating a license server, (2) accessing a database controlled by a license server; and (3) generating and printing reports on license usage.

The application 26 may not access the database 18 directly; rather, the request for a license is made through the licensing library 24 to the license server 20 over a path 28. Most network licensing schemes employ secured communication between the licensing library 24 and the license server 20. In contrast, the present invention uses the license access module (LAM) 27 the license library 24 and a plurality of license tokens to protect against the unauthorized use of software application in a computer network.

Referring once again to Figure 2, a license service binder 29 is shown coupled to the license server 20 over a path 30. The license service binder 29 is invoked by means known in the art, such as a network service program. The license service binder 29 locates all agents that are designated as servers on the network, and keeps track of which server is servicing which application. The license service binder 29 contacts each server on its table of available servers and requests a list of products it serves. Finally the license service binder 29 writes the contents of the table of available license servers and the list of products into a binding file 32 over a path 31. In Figure 2, the binding file 32 is coupled to the licensing library 24 over a path 33. The application 26

queries the binding file 32 to see which license server can service its request for a license.

A license production tool 34 is used by the software vendor to create a license token for transmittal to the network administrator. Receiving the license token, the network administrator installs it with the license administration tool 21 into the database 18 through license server 20.

II. License Token

Referring to Figure 3, the creation of a license token in a computer network employing the preferred embodiment of the present invention will be described. A computer network 38 is shown coupled with a license administration tool 39 and a single license server 44. The license server 44 communicates with a database 45. Applications 41, 42, and 43 are shown requesting licensing service from the license server 44. When a customer purchases a license for an application, such as a CAD/CAM program for its research and development department, the software vendor creates a license token with a license production tool, and delivers the license token to the customer's network administrator. A license token is a special bit pattern or packet representing a license to use a software application. The network administrator installs the license token 46 into the database of the license server using the license administration tool 39. Unlike the token used in a token ring which is passed from agent to agent, a license token in the preferred embodiment of the present invention is passed only between a license server and a licensing library for a predetermined amount of time. The predetermined amount of time corresponds to the time the license token is checked out of the license server. Currently, the license token is checked out to an application for no more than ten seconds, and the license token is returned as quickly as possible to the issuing license server. The license token 46 contains information encrypted in the vendor's format such as ,vendor identification, product and version numbers as well as the number of license units purchased

for the license token. A license unit corresponds to the license weighting for an agent connected to the computer network. For example, powerful workstations could require more license units to use a software application than an average personal computer.

The software vendor produces a license token using a license production tool 40. A path 47 illustrates how a license token 46' makes its way to a license administration tool 39 at the customer's site. There, the system administrator installs the license token 46' as license token 46 into the license database 45 of the license server 44. A path 48 indicates the transfer of the license token 46' from the license administration tool 39 to the license server 44 and into the database 45 as license token 46. The license server 44 is now ready to entertain requests from applications 41, 42, and 43 for a license to use the application corresponding to token 46 as well as other applications represented in its database 45.

It should be understood that each network may have a plurality of license servers and each license server may have in its database a plurality of license tokens for a variety of software applications. Referring again to Figure 3, if application A 41 requests and checks out the license token 46 for less than ten seconds, applications B and C 42, 43 would be unable to check out the license token 46 if their requests were made during the same time application 41 is checking out a license from the license token 46 because of the locking mechanism provided by database interface 19. Thus, to achieve concurrent license usage in network 38, it is preferred that the network administrator installs more than one license server. To minimize the task of recovering from license server crashes, it is also preferred that the system administrator spreads the license units for any one application among a plurality of strategically located license servers. For instance, if a network has four license servers, the network administrator may want to allocate the twenty license units for a particular popular application among four license tokens with

five license units for each license token. In the event one license server crashes or the license token is checked out, the other three license servers may provide licensing service to other applications.

Figure 4a illustrates the use of a license token to request for a license. As shown, a network 50 is provided, and is coupled to Applications 52, 54 and 56 respectively. Application 56 succeeded in requesting a license token from the license server 58 in step 59. The license token is transmitted to application 56 in step 60. When done, Application 56 returns the license token to the license server 58 in step 61. Aside from the license request function performed with the license token as shown in Figure 4a, the license token is also used in other critical stages of the licensing process. For example, an user may wish to run an application beyond the initial allocated time. As shown in Figure 4b, Application 68 makes a license renewal request 71 from the license server 70 with license token 72. Similarly, in Figure 4c the user makes a license release request 83 when the application no longer needs the license units. As such, the user updates the license token 84 by returning the updated license token to the license server 82 in step 85.

III. License Access Module

In Figure 2, a license access module (LAM) 27 is linked with the application 26 and the licensing library 24 to form the executable code that software vendors ship to the customers. The license access module 27 decodes and encodes the encrypted license token as it is being passed between the license server and the licensing library 24. Thus the level of security of an application from unauthorized use depends heavily upon how secure the license access module is.

Conventional network licensing schemes use public/private key encryption to encode sensitive information. Such a scheme is effective if the license server is in a trusted environment. However, the customer has the

same access to any agent in a network, including the license server. The security of the licensing scheme can be compromised by a user who decrypts the license server's private key. Once the unauthorized user determines the server's private key, he can decrypt all sensitive information on the license server. Should all license servers use the same key, as is frequently done, then all the security of the applications served by all the license servers will be compromised.

The license access module 27 first translates a license token from a vendor specific format to a format usable by the licensing library 24. The license access module accomplishes the translation in two modules. One module translates or decodes a license token from a vendor specific format to a licensing library format. The second module translates or encodes the updated license token from the licensing library format to the vendor specific format. The second module is invoked anytime the licensing library updates the information in a license token.

Upon receiving the license token in the licensing library format, the licensing library invokes routines which verify the correctness of the license by reviewing the following license information stored in the token: (1) flag, (2) maintenance contract date, (3) host name and domain, (4) product name, (5) host id number, (6) license serial number, and (7) expiration date of license. This is compared to the information maintained by the application. If the information matches, the license is verified. After completing the verification process, a routine in the licensing library is initiated which checks out the license by decrementing the license units in license token by the number of licensing units being checked out.

The decoding and encoding routines allow software vendors to implement their own security mechanism to protect their licenses from unauthorized use even though they reside at the customer's site.

Below is an example of a sample application using the licensing library and the license access module written in C language:

```
#define LIC_RENEWAL_TIME (60)              /set renewal time for this session/
#define EST_LIC_RENEWAL_TIME (LIC_RENEWAL_TIME x .9)

NL_vendor_id NL_Vendor_id = 1223;                    /set vendor #/
NL_prod_num NL_Prod_num = "02"                       /set product #/
NL_version NL_Version = ( 12/20/88, "1.0" );         /set version id #/

...
        status = NL_init (vendor_id, NULL, &job_id); /initialize license service/
        if (status != NL_NO_ERROR)      /accept job id if no error/
                {
                fprintf (stderr, "nl_init failed - error =
                %d\n", status );                     /error message if error and
                                                     return/
                return;
                }

        units = 3;
        code_funcs.encode_p = nl_encode;   /pointer to encode function/
        code_funcs.decode_p = nl_decode;   /pointer to decode function/
        if (signal (SIGALRM), alarm_intr ) == (void *) -1)      /set alarm if no
                                                                error/
                {
                perror ("Cannot set SIGALRM"); /otherwise, error message/
                return;
                }
        status = NL_request (job_id, NL_Prod_num,  /request a license/
        &NL_Version,
        units, LIC_RENEWAL_TIME, NL_L2_SRCH,
        &code_funcs, NULL,
        &req_handle, NULL, &app_info);

        if (status != NL_NO_ERROR)             /no error, license checked
                {                              out from license server/
                fprintf (stderr, "nl_request failed - error =
                %d\n", status);                /otherwise, error message/
                return;
                }

        /*
         * We got a license                    /license request successful/
         */

        alarm (EST_LIC_RENEWAL_TIME);          /set alarm for license renewal
...                                             time/
        Application Runs                       /runs application/
...
        status = NL_release (req_handle);      /request to release a license/
        if (status != NL_NO_ERROR)
                {
                fprintf (stderr, "nl_release failed - error =      /otherwise, error
```

```
                %d\n", status);                          messages/
            return;
            }

...
    int
    alarm_intr ()
        {

    status = NL_confirm (req_handle,      /renew licensing unit with
    LIC_RENEWAL_TIME, NULL);               licensing server/

    /*
     * Verify vendor private information
     */
        }

    If (status!= NL_NO_ERROR)
    fprintf (stderr, "nl_confirm failed - error =      /otherwise, error
    %\n", status);                                     message/
        {
            puts ("license renewed")                   /successful license
        }                                              renewal/
```

The sample application given above is accompanied by self-explanatory annotation to the right margin of the codes. Of particular interest are code_func.encode_p and code_func.decode_p. Encode_p and decode_p are pointers to the software vendor's encode and decode routines, respectively. Taking the pointers in the code_func variable, the licensing library can use the pointers to invoke the decoding and encoding routines in the license access module. The three major licensing library routines, request for a license (NL_request), release a license (NL_release) and renew a license (NL_confirm) invoke the decoding and encoding routines. For example of a license access module, see Appendix 1.

In implementing the license access module, the license server becomes merely a repository for license tokens. The licensing library coupled to the application performs the procedure of authenticating the license token prior to granting a license and therefore access to run the application.

Because the level of security of the system is dictated by the license access module, the software vendors are free to make the license access module as simple or as complex as they desire. In particular, they are free to

adopt any of the encryption schemes as part of their encryption routines. If the security mechanism is broken, and the encryption known to others, then the software vendors can easily remedy the situation by releasing a new version of the product with a new license access module.

```
/*
 * @(#)nl_lam.c 1.21 89/09/13
 * Copyright (c) 1988 by Sun Microsystems, Inc.
 */
/*
 ******************************************************************************
 *
 * Module: nl_lam.c
 *
 * Description:
 *      Contains the license access modules. Uses XDR to keep the license
 *      token in a machine independent form. Uses the des_crypt(3) routines
 *      to encrypt/decrypt the licensing information.
 *
 * Functions:
 *      nl_decode()     - Decodes a vendor's license token
 *      nl_encode()     - Encodes a vendor's license token
 *
 * Notes:
 *
 ******************************************************************************
 */
#include        <rpc/rpc.h>
#include        <string.h>
#include        <memory.h>
#include        "nl_types.h"
#include        "nl_prot_limits.h"
#include        "nl_request.h"
#include        "nl_token.h"
#include        "nl_vendor.h"
#include        "nl_licensel.h"
#include        "nl_errno.h"

/*
 ******************************************************************************
 * Module Local Definitions
 ******************************************************************************
 */
/*
 * Warning: This could be a potential security threat, leaving a unencoded
 * token lying around in memory.
 */
static  NL_license _nl_lic_tok;

/*
 ******************************************************************************
 * Function Declarations
 ******************************************************************************
 */
void            nl_lam();
static  int     _nl_decode();
static  int     _nl_encode();

/*
 ******************************************************************************
 *
```

```
 * Function: nl_lam
 *
 * Description:
 *      This function returns the address of the encode and decode routines.
 *
 * Input:
 *      conv_2_ascii_p  - storage that indicates whether to conv to ascii
 *      code_func_p     - storage contains address of encode/decode routines
 *
 * Output:
 *      conv_2_ascii_p  - returns an indication whether the token should
 *                        be converted to ascii so that it may be printable.
 *      code_func_p     - Contains address of encode/decode routines
 *
 * Returns:
 *      Nothing
 *
 * Notes:
 *      1) The reason that the conv_2_ascii_p value is returned is to
 *         inform the License Production Tool if it needs to do an
 *         ascii encoding on the license token. The license token must
 *         be in some type of ascii form to allow it to be transmitted over
 *         voice phone or to be Fax'ed to an end user.
 *
 ********************************************************************************
 */
void
nl_lam(conv_2_ascii_p, code_func_p)
bool_t          *conv_2_ascii_p;
NL_code_funcs   *code_func_p;
{
        if (conv_2_ascii_p != NULL)
        {
                *conv_2_ascii_p= TRUE;
        }
        code_func_p->decode_p = _nl_decode;
        code_func_p->encode_p = _nl_encode;
} /* nl_lam */

/*
 ********************************************************************************
 *
 * Function: nl_decode
 *
 * Description:
 *      Translates a license token from vendor specific format to a
 *      client library specific format.
 *
 * Input:
 *      en_tok_p        - Pointer to the encoded token
 *      en_length       - Length of the encoded token in bytes
 *
 * Output:
 *      tok_p           - Pointer to the token structure to fill in
 *      clnt_entry_p    - Pointer to the address of the first client usage
 *                        entry
```

```
 *
 * Returns:
 *      NL_NO_ERROR      - Decoding succeeded
 *      NL_E_DECODE_FAIL
 *
 * Side Effects:
 *      1) Allocates memory for the the client usage entries.
 *
 * Notes:
 *      1) If tok_p->vend_priv_p or tok_p->vend_priv_p->data_p is NULL
 *         then the call doesn't want it.
 *
 ********************************************************************************
 */
static int
_nl_decode(en_tok_p, en_length, tok_p, clnt_entry_p)
char            *en_tok_p;
int             en_length;
NL_token        *tok_p;
NL_clnt_entry   **clnt_entry_p;
{
        NL_usage_entry  usage_entry;
        NL_clnt_entry   *tmp_clnt_entry_p;
        XDR             xdrs;
        int             i;
        int             status;
        int             vend_size;

        /*
         * Make sure the size for the encoded token is divisible by
         * xdr bytes per unit.
         */
        if (en_length != RNDUP(en_length))
        {
                return (NL_E_DECODE_FAIL);
        }

        /*
         * Init an XDR stream
         */
        xdrmem_create(&xdrs, en_tok_p, (u_int)en_length, XDR_DECODE);

        status = NL_NO_ERROR;
        if (xdr_NL_license(&xdrs, &_nl_lic_tok) == FALSE)
        {
                xdr_destroy(&xdrs);
                return (NL_E_DECODE_FAIL);
        }

        if (_nl_lic_tok.magic == MY_VENDOR_MAGIC)
        {
                tok_p->vendor_id = _nl_lic_tok.vendor_id;
                (void) strncpy(tok_p->prod_num, _nl_lic_tok.prod_num,
                                NL_PROD_NUM_SIZE);
                tok_p->vers_secs = _nl_lic_tok.vers_secs;
                memcpy(tok_p->serial_num, _nl_lic_tok.serial_num,
```

```
                NL_SERIAL_NUM_SIZE);
        tok_p->expire_secs = _nl_lic_tok.expire_secs;
        tok_p->maint_secs = _nl_lic_tok.maint_secs;
        tok_p->total_units = _nl_lic_tok.total_units;
        tok_p->avail_units = _nl_lic_tok.avail_units;
        (void) strncpy(tok_p->host, _nl_lic_tok.host, NL_HOST_NAME_SIZE);
        (void) strncpy(tok_p->domain, _nl_lic_tok.domain,
                NL_DOMAIN_NAME_SIZE);
        tok_p->hostid = _nl_lic_tok.hostid;
        tok_p->flag = _nl_lic_tok.flag;
        tok_p->num_clnts = _nl_lic_tok.num_clnts;

        /*
         * Give the use a copy of the vendor private data if they
         * want it.
         */
        vend_size = MIN(_nl_lic_tok.vend_priv_size,
                NL_VEND_PRIV_SIZE);
        if (vend_size > 0 && tok_p->vend_priv_p != NULL &&
                tok_p->vend_priv_p->data_p != NULL)
        {
                tok_p->vend_priv_p->length = vend_size;
                (void) memcpy(tok_p->vend_priv_p->data_p,
                        _nl_lic_tok.vend_priv,
                        MIN(vend_size, tok_p->vend_priv_p->length));
        }

        tmp_clnt_entry_p = (NL_clnt_entry *)
                malloc((tok_p->num_clnts+1)* sizeof(NL_clnt_entry));
        *clnt_entry_p = tmp_clnt_entry_p;
        for (i=0; i < tok_p->num_clnts; i++, tmp_clnt_entry_p++)
        {
                xdr_NL_usage_entry(&xdrs, &usage_entry);
                tmp_clnt_entry_p->req_handle = usage_entry.req_handle;
                tmp_clnt_entry_p->units = usage_entry.units;
                tmp_clnt_entry_p->renew_secs = usage_entry.renew_secs;
                tmp_clnt_entry_p->clnt_secs = usage_entry.clnt_secs;
                tmp_clnt_entry_p->pid = usage_entry.pid;
        }
    }
    else
    {
        status = NL_E_DECODE_FAIL;
    }

    xdr_destroy(&xdrs);

    return (status);
} /* nl_decode */

/*
 ******************************************************************************
 *
 * Function: nl_encode
 *
 * Description:
```

```
 *      Translates a license token from a client library specific format
 *      to a vendor specific format. Verifies that the encrypted token
 *      does not exceed the NL_MAX_EN_TOKEN_SIZE.
 *
 * Input:
 *      tok_p           - Pointer to the token structure to read data from
 *      clnt_entry_p    - Pointer to the address of the first client usage
 *                        entry
 *
 * Output:
 *      en_tok_p        - Pointer to the encoded token
 *      en_length       - Length of the encoded token in bytes
 *
 * Returns:
 *      NL_NO_ERROR     - Encoding succeeded
 *      NL_E_ENCODE_FAIL
 *
 * Side Effects:
 *      1) Frees memory for the the client usage entries.
 *
 * Notes:
 *      1) If tok_p->vend_priv_p is NULL then the call doesn't want it.
 *
 ****************************************************************************
 */
static int
_nl_encode(tok_p, clnt_entry_p, en_tok_p, en_length_p)
NL_token        *tok_p;
NL_clnt_entry   *clnt_entry_p;
char            *en_tok_p;
int             *en_length_p;
{
        XDR             xdrs;
        NL_usage_entry  usage_entry;
        NL_clnt_entry   *tmp_clnt_entry_p;
        int             i;
        int             tok_len1;
        int             tok_len2;
        int             status;

        status = NL_NO_ERROR;

        /*
         * Init an XDR stream
         */
        xdrmem_create(&xdrs, en_tok_p, (u_int)NL_MAX_EN_TOKEN_SIZE, XDR_ENCODE);

        /*
         * If for some reason the license token that we will encode is
         * not the same as the previous one decoded (in our module
         * local global) then we copy the information in.
         */
        if (memcmp(_nl_lic_tok.serial_num, tok_p->serial_num,
                NL_SERIAL_NUM_SIZE) != 0)
        {
                _nl_lic_tok.magic = MY_VENDOR_MAGIC;
```

```
        _nl_lic_tok.vendor_id = tok_p->vendor_id;
        (void) strncpy(_nl_lic_tok.prod_num, tok_p->prod_num,
                NL_PROD_NUM_SIZE);
        _nl_lic_tok.vers_secs = tok_p->vers_secs;
        (void) memcpy(_nl_lic_tok.serial_num,tok_p->serial_num,
                NL_SERIAL_NUM_SIZE);
        _nl_lic_tok.expire_secs = tok_p->expire_secs;
        _nl_lic_tok.maint_secs = tok_p->maint_secs;
        (void) strncpy(_nl_lic_tok.host, tok_p->host, NL_HOST_NAME_SIZE);
        (void) strncpy(_nl_lic_tok.domain, tok_p->domain,
                NL_DOMAIN_NAME_SIZE);
        _nl_lic_tok.hostid = tok_p->hostid;
        _nl_lic_tok.flag = tok_p->flag;
        if (tok_p->vend_priv_p != NULL &&
                tok_p->vend_priv_p->data_p != NULL)
        {
                _nl_lic_tok.vend_priv_size =
                        MIN(tok_p->vend_priv_p->length,
                        NL_VEND_PRIV_SIZE);
                (void) memcpy(_nl_lic_tok.vend_priv,
                        tok_p->vend_priv_p->data_p,
                        _nl_lic_tok.vend_priv_size);
        }
        else
        {
                _nl_lic_tok.vend_priv_size = 0;
                _nl_lic_tok.vend_priv[0] = NUL;
        }
}
_nl_lic_tok.total_units = tok_p->total_units;
_nl_lic_tok.avail_units = tok_p->avail_units;
_nl_lic_tok.num_clnts = tok_p->num_clnts;

if (xdr_NL_license(&xdrs, &_nl_lic_tok) == FALSE)
{
        xdr_destroy(&xdrs);
        return (NL_E_ENCODE_FAIL);
}

if (clnt_entry_p != NULL)
{
        tmp_clnt_entry_p = (NL_clnt_entry *)
                malloc((_nl_lic_tok.num_clnts+1) *
                sizeof(NL_clnt_entry));
        tmp_clnt_entry_p = clnt_entry_p;
        tok_len1 = (int)xdr_getpos(&xdrs);
        for (i=0; i < tok_p->num_clnts; i++, tmp_clnt_entry_p++)
        {
                usage_entry.req_handle = tmp_clnt_entry_p->req_handle;
                usage_entry.units = tmp_clnt_entry_p->units;
                usage_entry.renew_secs = tmp_clnt_entry_p->renew_secs;
                usage_entry.clnt_secs = tmp_clnt_entry_p->clnt_secs;
                usage_entry.pid = tmp_clnt_entry_p->pid;
                xdr_NL_usage_entry(&xdrs, &usage_entry);

                /*
```

```
                        * Quick check here for the first pass of
                        * this loop. We find out how many bytes it
                        * took to encode 1 users entry. We then
                        * estimate the total number of bytes
                        * required and make sure that we will
                        * not exceed NL_MAX_EN_TOKEN_SIZE.
                        */
                       if (i == 0)
                       {
                               tok_len2 = (int)xdr_getpos(&xdrs) -
                                       tok_len1;
                               tok_len2 *= tok_p->num_clnts;
                               if ((tok_len1+tok_len2) >
                                       NL_MAX_EN_TOKEN_SIZE)
                               {
                                       i = tok_p->num_clnts;
                                       status = NL_E_ENCODE_FAIL;
                               }
                       }
               }
               free ((char *)clnt_entry_p);
       }
       *en_length_p = (int)xdr_getpos(&xdrs);

       xdr_destroy(&xdrs);

       return (status);
} /* nl_encode */
```

* C L M S *

CLAIMS

1. A method for controlling execution of an application program on a computer network, said method comprising the steps of:

storing a first execution authorization token in a first database by a first server, said first execution authorization token comprising authorization information of said application program and being stored in encrypted format;

retrieving said stored first execution authorization token for said application program by said first server, said retrieval being requested of said first server by an authorization routine of an authorization library for authorizing said application program for execution;

attempting to verify said authorization information stored in said retrieved first execution authorization token against authorization information maintained by said application program in conjunction with said authorization routine after said retrieved first execution authorization token has been decrypted by an application program specific execution access module integrated with said application program;

if said authorization information stored in said retrieved first execution authorization token cannot be verified against said authorization information maintained by said application program, halting execution of said application program;

if said authorization information stored in said retrieved first execution authorization token can be verified against said authorization information maintained by said application program,

(i) authorizing said application program for continued execution upon said verification by said authorization routine;

(ii) updating said retrieved first execution authorization token upon said authorizing by said authorization routine; and

(iii) returning said updated first execution authorization token by said authorization routine to said first server for storage in said first database after said updated first execution authorization token has been re-encrypted by said application program specific execution access module integrated with said application program.

2. The method as defined in claim 1, wherein said re-encrypted first execution authorization token is returned by said authorization routine to said first server for storage in said first database within a predetermined time period.

3. The method as defined in claim 1, wherein said method further comprises the step of receiving said first execution authorization token from an authorization entity for said application program and providing said first execution authorization token to said first server for storage in said first database,

said first execution authorization token being received from a distribution medium of said application program and provided to said first server in encrypted format by an authorization administration tool.

4. The method as defined in claim 3, wherein said method further comprises the step of producing said first execution authorization token on said distribution medium in said encrypted format by an authorization entity for said application program by an authorization production tool.

5. The method as defined in claim 1, wherein said

* C L M S *

method further comprises the step of receiving said application program integrated with said application program specific execution access module from a distribution medium of said application program and installed onto said computer:

said application program specific execution access module being integrated with said application program and placed on said distribution medium by said authorization entity for said application program.

6. The method as defined in claim 1, wherein said application program specific execution access module comprises a decrypting module for decrypting said first execution authorization token and an encrypting module for encrypting said first execution authorization token, said decrypting and encrypting module being executed by said computer.

7. The method as defined in claim 1, wherein said step of updating said retrieved first execution authorization token upon said authorizing comprises decrementing execution authorization units in said retrieved first execution authorization token.

8. The method as defined in claim 1, wherein said method further comprises the steps of:

retrieving said stored first execution authorization token for said application program by said first server, said retrieval being requested of said first server by an unauthorizing routine of an authorization library for unauthorizing said application program for execution;

verifying said authorization information stored in said retrieved first execution authorization token against authorization information maintained by said

application program by said unauthorizing routine after said retrieved first execution authorization token has been decrypted by said application program by an application program specific authorization access module integrated with said application program;

unauthorizing said application program for execution upon said verification by said unauthorizing routine;

updating said retrieved first execution authorization token upon said unauthorizing by said unauthorizing routine; and

returning said updated first execution authorization token by said unauthorizing routine to said first server for storage in said first database after said first execution authorization token has been re-encrypted by said application program specific execution access module integrated with said application program.

9. The method as claimed in claim 1, wherein said method further comprises the steps of:

storing a second execution authorization token in a second database by a second server, said second execution authorization token comprising authorization information of said application program and being stored in encrypted format;

storing first and second server identifiers identifying said first and second servers in a binding file, said authorization library accessing said binding file to select said first server to retrieve said first execution authorization token; and

identifying said first and second servers and storing said first and second server identifiers into said binding file by an authorization service binder.

10. The method as defined in claim 9, wherein said method further comprises the step of receiving said first

and second execution authorization tokens from a distribution medium of said application program and providing said first and second execution authorization tokens to said first and second servers for storage in said first and second databases respectively by an authorization administration tool, said first and second execution authorization tokens being received by said authorization administration tool and provided to said first and second servers in encrypted format.

11. The method as defined in claim 10, wherein said method further comprises the step of producing said first and second execution authorization tokens on said distribution medium in said encrypted format by an authorization entity for said application program by an authorization production tool.

12. An apparatus for controlling execution of an application program, said apparatus comprising:

first database means comprising a first database for storing a first execution authorization token, said first execution authorization token comprising authorization information of said application program and being stored in encrypted format;

first server means coupled to a computer and said first database means comprising a first server for storing said first execution authorization token into said first database and for retrieving said stored first execution authorization token from said first database;

an authorization library comprising an authorizing routine for requesting said first server to retrieve said first execution authorization token for said application program, attempting to verify said authorization information stored in said retrieved first execution authorization token against authorization information maintained by said application program after

* C L M S *

said retrieved first execution authorization token has been decrypted by said application program, if said authorization information stored in said retrieved first execution authorization token cannot be verified against said authorization information maintained by said application program, said authorizing routine halting said execution of said application program and if said authorization information stored in said retrieved first execution authorization token can be verified against said authorization information maintained by said application program, said authorizing routine (i) authorizing said application program for execution upon said verification, (ii) updating said retrieved first execution authorization token upon said authorizing, and (iii) returning said updated first executed authorization token to said first server for storage in said first database after said updated first execution authorization token has been re-encrypted by said application program, said authorization library including said authorizing routine being executed by said computer; and

an execution access module specific for said application program and integrated with said application program for decrypting said retrieved first execution authorization token and re-encrypting said updated first execution authorization token, said execution access module being executed by said computer.

13. The apparatus as defined in claim 12, wherein said re-encrypted first execution authorization token is returned by said authorizing routine to said first server for storage in said first database within a predetermined time period.

14. The apparatus as defined in claim 12, wherein said apparatus further comprises authorization administration tool means coupled to said first server

means comprising an authorization administration tool for initially receiving said first execution authorization token from a distribution medium of said application program and providing said first execution authorization token to said first server for storage in said first database, said first execution authorization token being received by said authorization administration tool and provided to said first server in encrypted format.

15. The apparatus as defined in claim 14, wherein said apparatus further comprises an authorization production tool means comprising an authorization production tool for producing said first execution authorization token on said distribution medium in said encrypted format by an authorization entity for said application program.

16. The apparatus as defined in claim 12, wherein said application program integrated with said application program specific execution access module is received from a distribution medium of said application program specific execution access module being integrated with said application program and placed on said distribution medium by said authorization entity for said application program.

17. The apparatus as defined in claim 12, wherein said application program specific execution access module comprises a decrypting module for decrypting said first execution authorization token and an encrypting module for encrypting said first execution authorization token, said decrypting and encrypting modules being executed by said computer.

18. The apparatus as defined in claim 12, wherein said authorizing routine performs said update of said

retrieved first execution authorization token upon said authorizing by decrementing execution authorization units in said retrieved first execution authorization token.

19. The apparatus as defined in claim 12, wherein said authorization library further comprises an unauthorizing routine for requesting said first server to retrieve said first execution authorization token for said application program, verifying said authorization information stored in said retrieved first execution authorization token against authorization information maintained by said application program after said retrieved first execution authorization token has been decrypted by said application program, unauthorizing said application program for execution upon said verification, updating said retrieved first execution authorization token upon said unauthorizing, returning said updated first execution authorization token to said first server for storage in said first database after said updated first execution authorization token has been re-encrypted by said application program.

20. The apparatus as defined in claim 12, wherein said apparatus further comprises:

second database means comprising a second database for storing a second execution authorization token, said second execution authorization token comprising authorization information of said application program and being stored in encrypted format;

second server means coupled to said computer and second database means comprising a second server for storing said second execution authorization token into said second database and for retrieving said stored second execution authorization token from said second database;

binding file means coupled to said computer

comprising a binding file for storing first and second server identifiers identifying said first and second servers, said authorization library accessing said binding file to select said first server to retrieve said first execution authorization token; and

authorization binding means coupled to said first and second server means and said binding file means comprising an authorization service binder for identifying said first and second servers and storing said first and second server identifiers into said binding file.

21. The apparatus as defined in claim 20, wherein said apparatus further comprises authorization administration tool means coupled to said first and second server means comprising an authorization administration tool for initially receiving said first and second execution authorization tokens from a distribution medium of said application program and providing said first and second executions authorization tokens to said first and second servers for storage in said first and second databases respectively, said first and second execution authorization tokens being received by said authorization administration tool and provided to said first and second servers in encrypted format.

22. The apparatus as defined in claim 21, wherein said apparatus further comprises an authorization production tool means comprising an authorization production tool for producing said first and second execution authorization tokens on said distribution medium in said encrypted format by an authorization entity for said application program.