License Manager Design Document

Craig Fields

June 4, 1992

TO DO

Note – This is not a final draft. Changes to come include:

- A rewrite of chapter 2, License Manager Configuration, including introduction of regular expressions for matching, and how they are used in hostnames.
- Explanation in section 3.3.2, Finding the Master License Server, of a non-Hesiod mechanism for finding the license servers.
- Changes in the server-server protocol removing the exchange between servers of specific details on licenses granted, as this information is not required; only details on the number of active licenses need be distributed.
- Settling of values for the timing parameters in 3.4.3.
- Filling out the Administration Protocol, 3.5.
- Many other unmentioned details...

Please send flames, comments, or suggestions on this design or document to cfields@mit.edu.

Contents

Chapter 1

Requirements

A license management system is a general mechanism for restricting the use of applications. On startup, an application queries a license server for permission to run. The server may grant or deny permission on the basis of any information it has available to it, and then the application will act accordingly.

The main feature a license management system can provide in terms of access restrictions that other mechanisms cannot is the ability to restrict access on the basis of how many people currently have access. This is the primary motivation for using a license management system. Without such a system, if you want to be able to run an application from the network on any of your machines, you need to buy a site license. With such a system, you can run an application anywhere you want, and only pay for some limited number of copies you expect to need to run concurrently.

Some vendors ship their applications with license servers, so we can buy software from them in terms of numbers of concurrent licenses. However, as we expect to buy software from many different vendors, we may wind up with as many different kinds of license servers. This is a potential headache for three reasons. First, we will need to maintain all of these servers. Second, different servers may be incompatible, and may therefore need to be run on different hosts, increasing the number of hosts we need to maintain for license services. Finally, different vendor license servers may not be suitable to our needs in terms of things like access restriction and redundancy.

Furthermore, other vendors do not yet have license server technology, but we would still like to license their products by number concurrent users rather than by site.

For these reasons, it is important for us to have a license server distinct from those provided by our software vendors. The idea here is that vendors without license servers will trust us sufficiently to use our technology with their product in order to incorporate concurrent license restrictions into their product, allowing us the kind of licensing we desire. Further, we hope to convince other vendors with license server technology to offer us their product without it, so that we can use ours instead. The result of these efforts would be a more homogeneous system, easier to maintain, which better meets our needs.

Applications generally have license server support compiled in. Since companies tend to be protective of their source code, or might be unwilling to do the work to incorporate our technology into their product, we need a solution that is independent of the application we wish to license. This brings us to our first requirement:

- There should be a generic method of taking vendor supplied software and making it work with the license server.

There are other technical requirements, which should be true of most services in our network environment:

- The license server client should be implementable on microcomputer platforms as well as workstations.
- The license server system should scale, being able to service thousands of clients.
- The service should be reliable, and therefore offered in a redundant fashion.

From the motivation for having a license server, we begin the list of types of restriction the license server should be able to do.

- The license server should be able to restrict software usage to a limited number of concurrent users.

This kind of restriction reduces your resources (as measured in number of applications available to run) from those you have with a site license, and so you need ways of allocating them. For instance, a piece of software may be intended (or licensed) for use only by a subset of the user community, such as students taking a particular class, or members of a research group.

- The license server should be able to restrict software usage to specified subsets of the user community.

Software bought by an organization is probably intended only for use in that organization. Thus, the license server needs to restrict use to subsets of machines on the Internet (machines on campus, machines in a particular laboratory, etc.)

- The license server should be able to restrict software usage to a particular set of host machines.

There are some important requirements from the user's standpoint. First of all, when all licenses are in use, it is not nice to force the user to periodically retry running the application until a license is available.

- The license server should have a facility for queueing users when all licenses are in use, and informing them when a license is available for their use.

Secondly, it is unkind to boot the user out of the application when contact with the license server is lost. Furthermore, in our environment this is untenable, since the user may not be able to save his files at that time for the same reason contact with the license server has been lost.

- The license server should not be anti-social. Users should not be hurt in any way by the license system when their applications lose server contact.

There are also a few requirements from an administration standpoint. Application usage statistics should be available for various purposes, including evaluating the demand for a particular application to determine if we have a reasonable number of licenses with which to satisfy it.

- The license server should be able to provide information on application usage, like how many licenses are out at a particular time and how long the queue is.

As the amount of software using the license server grows, it will become important for the people responsible for it to be able to change the restrictions placed on it (such as number of licenses, hosts allowed to run the software, etc.).

- There should be a simple administration interface to the server, allowing people responsible for the software to change the conditions of limitation easily.

Chapter 2

License Manager Configuration

The license server reads license description data from files when it starts up. Changes to these files are managed by a separate system to keep the complexity away from the license server.

For each software package, there is a file describing the various aspects of the licenses. First, there may be several pools of licenses for a given package. Each pool may have its own restrictions. For instance, the first pool for a given package may express the fact that fifty licenses are reserved for use by a particular class of users. The second pool could then say that the remaining two hundred licenses can be used by anyone.

For each pool, license use may be restricted on the basis of platform, host, or user. For each, the restriction may be specified by lists of included and excluded sets. What can be done is most easily shown by random example.

```
POOL
  LICENSES 50
  PLATFORMS decmips/2 vax/1
  HOSTS eecs foo.mit.edu bar.mit.edu
  USERS course6 -joehacker
POOL
  LICENSES 100
  PLATFORMS decmips/2 vax/1
  HOSTS *.mit.edu
  USERS * -joehacker
  ACCESSMSG Go away Joe.
```

The first pool of licenses reserves 50 licenses for use on hosts in the eecs group, foo, and bar. To run on a decmips takes two licenses; to run on a vax takes one. The user must be in the course6 group and not be joehacker. If the requester does not satisfy the requirements for the pool, the next pool will be tried. The second pool contains the remaining 100 licenses, and allows anyone in *.mit.edu have the licenses execept for joehacker. A message is returned to those who are denied access, specified by ACCESSMSG.

The "groups" such as course6 and eecs (containing users and hosts, respectively) comes from a merge of the Moira database and a specially managed license-server database. The latter is used both for data that needs to be changed in real time and for data that has no purpose in cluttering the Moira database.

There will be other information in the file as well, such as a pointer to where to get any application specific bits that need to be sent to the client when a license is granted.

Chapter 3

License Manager Communications

3.1 Overview

The license management system consists of a set of one or more servers, of which normally only one is a master license server and the rest are backup servers. A master license server does the work of granting licenses and keeping track of how many licenses are checked out, while a backup license server stands ready to become a master should it lose contact with its master; the master keeps its backups' license databases up-to-date should any of them need to become masters.

Figure 3.1 depicts a set of three servers running in a normally operating network. Solid lines indicate network connectivity between hosts. Since all hosts are in contact here, there is only one master. Figure 3.2 depicts the same three servers after a network partition has occurred. Dotted lines indicate a lack of network connectivity between hosts. There is one master for each network partition; this should always be true. Whatever happens to the network, the servers will notice what is going on and adapt accordingly.

A client, once it is in contact with a master license server, is responsible for maintaining contact with the license management system. If a client loses contact with its master, it should actively seek contact with another master until it makes contact or it exits. Master servers make the assumption that the client behaves in this way, and so never investigate a timed-out client.

A specific type of authentication between clients and servers is not required by the protocol. Instead, the protocol uses generic authentication records, so that many forms of authentication

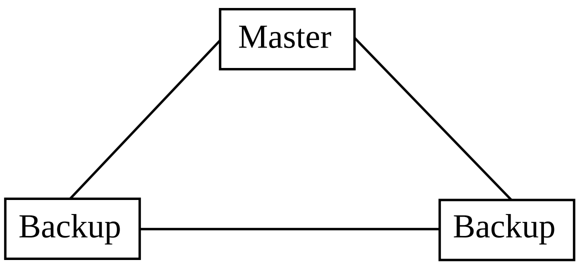

Figure 3.1: Servers in a normally functioning network

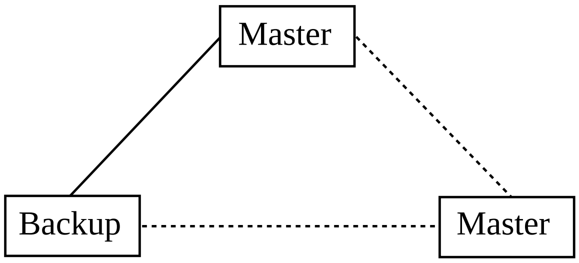

Figure 3.2: Servers in a partitioned network

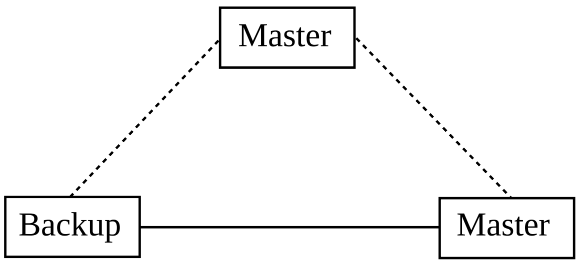

Figure 3.3: Another partitioned network

may be used. However, only one type of authentication, Kerberos, is referenced in this description.

3.1.1 Client Operation

Client Startup

When a client starts up, it first asks Hesiod what hosts are license servers. As any one of the machines Hesiod returns could be the master license server, and the authentication protocol may need to know the identity of the master host, the client must ask one of servers which of them is master. The one it asks should be selected randomly, and if it doesn't get a response after a time T_{poll} should try another host at random with the same timeout until it gets a response. It should make the request of each host at most N_{poll} times. This process will be referred to as the *find master* procedure, as it is used in other instances as well. (Note: N values represent how many times something should be done. T values specify time intervals for retries or timeout values, in seconds. Values are proposed in a later section.) If *find master* procedure fails to locate a master, the client should return to the user an appropriate error message.

Knowing the identity of the master, the client may proceed with master-identity dependent authentication procedures (if any), and contact the master with an authenticated license request. If no reply is returned after a time T_{retry}, the client should resend the request up to N_{retry} times (subject to authenticator validity constraints). If this cycle fails, the client should fall back to the *find master* procedure, exiting if no master is found, trying a license request again if a master is found.

The client will receive from the master either an error reply or a license grant reply. The license grant reply may be of one of two types: it may be either a current license (the client has permission to run now), or a future license (the client has been placed in the queue, to receive permission to run at some time in the future).

In the case of a current license, the client may need to request application specific information from the server to be able to run. Once again, it should use the T_{retry} and N_{retry} parameters for getting a response from the server. If the server fails to respond, the client should use the *find master* procedure. In the case of no response, it should exit with an appropriate error message. For success, it should ask the (potentially new) master for the application information. Authentication procedures should provide for the new master trusting the word of the client that it has been granted a license by a different master, as described later.

Maintenance of Client-Server Contact

Once the application is running, or while it is waiting for its future license to become a current license, the client should be testing the connection with the server at a T_{ping} interval. If the server fails to receive N_{MC} consecutive pings, it will consider the client to be dead, and free the license for use by someone else.

The server should reply to the client pings. If the client fails to receive N_{CM} consecutive replies, it should consider the master out of reach and use the *find master* procedure to locate a new master to ping. It should only give up finding the master in the case that the application exits. Once it finds a new master, it should begin pinging it. The new master may not have full information about the client, and request that information of the client.

It may also happen that a new server appears within reach on the network. That server may also be a master, and the two masters will decide between them which should become the single master. In this case, a client may have to change masters. This is done by the client's former master returning an error to a ping request rather than returning a ping reply. This error message will tell the client what server to switch to as master.

3.1.2 Server Operation

License Granting Behavior

The license granting behavior of master servers is fairly straightforward. If a master is in contact with all of the servers, then it knows that it is the only master. Because it is the only master, it is allowed to grant all of the licenses.

When a master loses contact with any of its backups (and it does not know that the backup has crashed) it assumes that the network has been partitioned into two pieces, as this is the most likely case. This assumption has several consequences.

First, all of the backups it has lost contact with are now in their own partition, and will establish a new master. The new master will not be in contact with all of the servers (namely, at least, the original master) and make the same assumption that there is one other master granting licenses. Since there are two masters granting licenses, each should only grant half of the remaining free licenses in order to prevent overgranting. For a time T_{split} after the network partition, licenses may not be returned to the free pool due to timeouts, since the clients with said licenses will only be able to ping one of the masters. However, a master may timeout a client with which it has had contact since the partition.

Additionally, a master may notice it has lost contact with all clients on a particular subnet. This will also be assumed to be the result of a network partition, with the result that clients on that subnet may similarly not be timed out for a time T_{split}.

Establishment of Master Servers

Changes in network connectivity may cause subsets of servers to establish or lose contact with one another, but at all times there should be at most one server operating as a master within a single network partition. There are three cases from a server's point of view where its state, whether it is a master or backup, might change. First there is the case of a master making contact with another master. Second, there is the case of a backup losing contact with its master. Finally, there is the case of a server booting up that needs to determine whether it should become a backup or a master.

The database update message (in conjunction with its corresponding reply), which all masters send out periodically (frequency T_{update}), is the mechanism by which contact between servers is maintained.

The case of two masters making contact (when one master receives the update message of another) can only happen in the case where two network partitions merge, as in a network changing from Figure 3.2 to Figure 3.1. When this happens, a master enters a *frozen master* mode where its data base state, as indicated by its update messages, appears to freeze. In freezing their states as seen by the outside world, multiple masters after a short time will all have a consistent picture of the world. With this picture, all of the masters can decide based on the same data which server should remain a master with assurance that the other masters will have made the same decision.

Note that during the time a master is in "frozen" mode, its database may still actually change; it is allowed to grant licenses as it was prior to the freeze.

Any particular server may be in the frozen state for a maximum of N_{frozen} update messages. It will exit the frozen state sooner than this (and enter backup mode) if it decides from another server's update message that that server should become the master. If in this time it doesn't hear from a smarter server, it will become the master itself.

A backup is considered to have lost contact with its master when it misses N_{BM} update messages. When this happens, the backup enters a frozen master mode, which is different from the type masters enter in that the backup is not allowed to grant licenses. It does, however, send out update messages. During the time they are in frozen master mode, the backups will learn about each other, and just like masters in this mode, will determine which of them is to become master.

A server at startup time behaves like a backup which has lost its master.

License Database Consistency

Master servers keep backup servers up to date by using update messages. Every T_{update} seconds, the master sends out any changes that have occurred to the database since the last update. Included in the update message are the database version number which the changes amend, and the version number the changes bring the database up to. The backups reply with their database version numbers.

The backups do not apply the changes if their own version number does not match the version which the changes are to amend, and in this case reply with their older database version number. In this way, the master knows when a backup has an out of date database (due to dropped packets or other reasons), and can either retransmit all changes necessary to bring the backup up to date, or dump the entire database.

When multiple masters connect and select a master among themselves, the new master needs to integrate the database information from the rest into its own database, and then dump the new database to the backups. For this purpose, there is a brain-dump request a master can make of a backup. It obtains the necessary information from the former masters this way, merges databases internally, then sends out update messages to the backups to bring them up to date.

3.2 General Inter-Process Protocol

Following is a general description of the protocol.

All messages in the protocol have the following form:

short	Protocol version
short	Packet type
long	Packet length
	Body

The content of the body is determined by the protocol version and packet type. In general, requests are "acked" by their replies. If the reply not received for some reason, the request is retransmitted. Exceptions to this rule are noted in the detailed protocol description.

Protocol messages are described in a standard format. That format is:

Type	the type of protocol message
Context	what context this message is used in, e.g. A $\rightarrow$ B, where A, B may be any of:

Client	a potential licensee
Master	a master server of licenses
Backup	a backup server of licenses
Server	either kind of server
Any	any kind of server or client
Administrator	administration client

Authentication	Whether or not this message is authenticated. An authentication record is appended to the other data in the packet.
Content	The content of the message not including the message header.

3.3 Client-Server Protocol

3.3.1 Records Used in Client-Server Messages

Authentication Protocols

The authentication protocols that may be used are fairly independent of the license server protocol. The authentication protocol to be used with the client is chosen in the initial license granting messages. Servers are assumed to always use the same authentication protocol amongst themselves. An authentication protocol can involve using different types of authentication records in different messages. For instance, the first message from client to master may contain setup information for the authentication system.

Null	The ever popular empty protocol
Kerberos-S1	The first version of the Kerberos inter-server authentication protocol
Kerberos-C1	The first version of the Kerberos client-server authentication protocol

Authentication Record

short	Authentication Protocol
short	Record type for protocol
short	Length of authentication record
	Protocol/type dependent data

Protocol Support Record

shorts	Protocol versions supported
shorts	Authentication protocols supported

Protocol Parameters Record

It may be desirable to change different timing characteristics of the client verification protocol without needing to recompile all of the clients. Therefore, the server provides this information.

The specific details of the client verification protocols are discussed in a later section.

short	Server timeout period (T_{CM})
	Servers to ping on timeout
short	Frequency of ping (T_{freq})

Package Identification Record

The *Package Identification Record* identifies what package a license request or license pertains to.

string	Package name
string	Package version

User Identification Record

We may want to support more than one user identification record type. Currently, this consists of the Kerberos triple:

principal
instance
realm

Machine Identification Record

The *Machine Identification Record* is mainly used to determine for what machine type a particular package is being requested, in order to send the appropriate package-dependent bits. This information may also be used for access control, along with the *User Identification Record.*

string	Machine type
string	OS type
string	Machine name
long	Machine address

Access Description Record

The *Access Description Record* specifies exactly what is being requested, and for whom. It additionally specifies whether a future access grant (entry in the queue) is acceptable or not. When used as part of an *Access Grant Record*, this access type field specifies whether the grant was for the present or the future.

User Identification Record
Machine Identification Record
Package Identification Record
Access type: Current or Future

Access Grant Record

Access Description Record
Timestamp (time of checkout)
Authenticator (authentication protocol dependent)

Bits Record

Sequence of

offset	offset to place bits
length	length of bits (in bytes)
1's and 0's	the bits

3.3.2 Client-Server Message Descriptions

General Error Messages

An *Error* message gives information about why a request could not be satisfied. An error response of "Protocol violation" is possible in cases where the client tries to do something that is just not ever allowed; for instance, asking a backup to supply bits.

Type:	Error
Context:	Server $\rightarrow$ Client
Authentication:	No
Content:	Type of packet this is a response to
	Serial number of packet this is a response to
	Error code
	Error code dependent data
	Text the client should give to the user (i.e., for access failure, who to contact for information)

Finding the Master License Server

When a client starts up and gets a list of servers, it doesn't know which one is the master. Thus, there is an *Orientation Request* which the client may pass to any server to find out who is master. Note that the client passes an entire *License Request Record* to the server. This allows the potential functionality of allowing different servers to be masters for different packages, or to handle different protocols.

Type:	Orientation Request
Context:	Client $\rightarrow$ Server
Authentication:	No
Content:	Access Description Record
Errors:	Package unknown
	Package disabled
	Protocol version not supported
	Authentication Class not supported

Type:	Orientation Response
Context:	Server $\rightarrow$ Client
Authentication:	No
Content:	Internet address
	Port number

Acquiring Permission to Run

Once the client has determined who the master is (and possibly done preparatory work for authentication), it has the information it needs to send a request to the master for a license. If the user/machine combination has sufficient permission, a license will always be granted. The license may be of one of two forms: a current license, allowing use of the resource now, or a future license (i.e., the user is placed in a queue).

Type:	Access (license) Request
Context:	Client → Master
Authentication:	Yes
Content:	Access description record
	Protocol support record
Errors:	Protocol not supported
	Access denied (by user, group, machine)
	Maximum licenses already checked out by user
	Package unknown
	Package disabled
	Authentication failed
	Unrecognized address type
	Not Master (and who is)

Type:	Access Grant
Context:	Master → Client
Authentication:	Yes
Content:	Access grant record
	Protocol parameters record
	Queue position

Obtaining Application Data

Once the client has a license, it can use that to get the magic bits required to actually run the application. These magic bits may be missing parts of the application, or a key to decrypt part or all of the application.

Type:	Bits Request
Context:	Client → Master
Authentication:	Yes
Content:	Access grant record
Errors:	Couldn't read bits
	Grant record checksum failed
	Grant record timestamp expired

Type:	Bits
Context:	Master → Client
Authentication:	Yes
Content:	Bits record

Promotion of a Future License to a Current License

In the case where a client's *Access Request* resulted in a future license rather than a current license, the client needs to be informed when its access is promoted to current. This is done via an *Access Upgrade* message from the master. The *Access Upgrade* message contains a new *Access Grant Record* which the client may use in a subsequent *Bits Request.* The old *Access Grant Record* should be thrown away. It also contains a new *Protocol Parameters Record*, as the timing for network connection verification may be different.

Type:	Access Upgrade
Context:	Master → Client
Authentication:	Yes
Content:	Access grant record Protocol parameters record

Network Connection Verification

When a client-server connection has been established, it is periodically verified by *Ping* messages.

Type:	Ping Query
Context:	Client → Master
Authentication:	No
Content:	The client's concept of whether it has a current or future license.

Type:	Ping
Context:	Master → Client
Authentication:	No
Content:	The Master's concept of whether the client has a current or future license, including queue state if applicable.
Errors:	Not master Database inconsistent (send Access Grant Record)

Request for Access Grant Record

Sometimes, the client may not receive its *Access Upgrade* message, and notice via a *Ping* message that it has been promoted. This message provides a way for the client to request the upgrade be resent. Conversely, the client may have lost contact with its server, and begun sending ping requests to all of the servers. In this case, the server may wish to verify the identity of the client (or update its database) by requesting a copy of the client's access grant record.

Type:	Access Grant Record Request
Context:	Client → Master or Master → Client
Authentication:	No
Content:	None

Type:	Access Grant Record Reply
Context:	Master → Client or Client → Master
Authentication:	No
Content:	Access Grant Record

Releasing a License

When a user quits an application, or no longer desires to wait to come to the head of the queue, the client should tell the server to release the license. If this fails to happen, the license will not become available until the client-server connection times out. Because this is rather important to the expediency of granting licenses, this message is acked.

Type:	Access Release Request
Context:	Client → Master
Authentication:	Yes
Content:	Access grant record

Type:	Access Release Acknowledge
Context:	Master → Client
Authentication:	Yes
Content:	The checksum from the released record

Authentication Messages

Some forms of authentication may require more initialization messages between hosts than is provided for in the rest of the protocol. Thus the protocol provides a generic authentication message to be used for these purposes. This message contains only an authentication record of the type being used for authentication.

Type:	Authentication Message
Context:	Any → Any
Authentication:	Yes
Content:	None

3.4 Server-Server Protocol

3.4.1 Records Used in Server-Server Messages

Global Database ID/Status

8 bits	State information (current/future, frozen, bits sent)
byte	Index of granting host
short	Serial number of grant on that host

License Identifier

byte	Number of licenses required
3 bytes	License index

Database Record

long	Global ID/Status
long	IP Address
long	License Identifier

3.4.2 Server-Server Message Descriptions

Master State Message

The master will send out a Master State message every T_{update} seconds regardless of changes to the database. This message serves two purposes: First, the backups know the master is up. Second, it enables the backups to keep their internal databases up-to-date. This message may contain the entire master database in the case of a merge operation.

Type:	Update
Context:	Master → Backup
Authentication:	Yes
Content:	Active size of database Database version this updates from Database version this updates to Number records following Records

Type:	Update Reply
Context:	Backup → Master
Authentication:	No
Content:	Latest database version

Brain Dumps

Type:	Brain Dump Request
Context:	Master → Backup
Authentication:	Yes
Content:	Starting database version (0 for beginning) Ending database version (-1 for most recent)

Type:	Brain Dump Reply
Context:	Backup → Master
Authentication:	Yes
Content:	Starting database version Ending database version Number of records following Records

3.4.3 Timing Parameters

T_{poll}	During client startup, the timeout period for a response from a orientation request (1 second)
N_{poll}	During client startup, the number of times each master should be polled before giving up (10)
T_{retry}	General timeout value for server requests
N_{retry}	Number of retries allowed for server requests

T_{ping}	Frequency with which client maintains contact with server (180 seconds)
N_{MC}	Number of consecutive pings the server misses before timing out the client (3)
N_{CM}	Number of consecutive ping replies the client misses before seeking a new server (3)
T_{update}	Frequency with which master sends update messages (60 seconds)
N_{BM}	Number of consecutive updates backup misses before entering frozen mode (3)
N_{MB}	Number of consecutive update replies master misses before assuming backup is unreachable (3)
N_{frozen}	Number of update cycles the frozen state lasts (3)
T_{split}	Time after a network partitioning that clients may be timed out (3600 seconds)

Relations:

$T_{ping} \times N_{CM} > T_{update} \times (N_{BM} + N_{frozen})$ so that when the client times out the master and goes to seek a new master, the backups already have the problem in hand.

3.5 Administration Protocol

3.5.1 Learning Server State

Type:	State Request
Context:	Administrator $\rightarrow$ Server
Authentication:	No
Content:	None

Type:	State Reply
Context:	Server $\rightarrow$ Administrator
Authentication:	No
Content:	Various interesting state information about the server

3.5.2 Forcing License Grants

The server will grant package access to the user when it is requested (one time only), regardless of whether or not the user has access. If the record says that queueing is undesirable, the license is granted regardless of whether or not there are any available; otherwise, the user is placed in the queue.

Type:	Grant License
Context:	Administrator $\rightarrow$ Master
Authentication:	Yes
Content:	Access Description Record
Errors:	Same as Access Request
	No Error (for an ack)

Chapter 4

Application Wrappers

There are three properties an application wrapper should have:

- It should be easy to wrap around any application.
- It should allow the application to run only after getting permission from a license server, and thereafter keep in contact with the server as long as the application is running.
- While maintaining the previous two properties, it should make it reasonably difficult for the user to defeat the system and run the application anyway.

The first property implies that the wrapper isn't going to be doing any modification of the application binary or behavior. The second is just a description of the way anything running under a license server should behave. The third implies that the raw application should not be left lying around where the user can just run it, bypassing the licensing system.

The general idea for satisfying the third property is to have most of the application binary available as it would be normally for a given platform (whether on local disk or through a network filsystem), with critical blocks missing. When the wrapper starts, it gets permission from the server to run, and then somehow supplies the complete application to the system to be run, without it falling onto a filsystem where the user could grab it.

The last bit is the trick, and in certain cases it might make sense to settle for the application being on disk for only a short period of time (i.e., BSD Unix). Following are reasonably simple ways to do this:

BSD	Write the application to local disk, including the extra bits obtained from the license server, and vfork. The child executes the binary, while the parent unlinks it as soon as its child has started executing.
AIX, SunOS, Ultrix	Write the application to local disk, not including the extra bits, and fork. The child requests tracing by its parent, then execs the binary. When the child is loaded, the parent writes the extra bits into its image, and then detaches from the child to let it run independently.
Mac's and PC's	The operating systems aren't as restrictive here, so you can pretty much do whatever you wish.