On the Design of Bouncer: A Robust and Flexible License Management Service for Avoiding Illegal Use of Software

[1]*Francisco Vilar Brasileiro*
fubica@dsc.ufpb.br

[1,2]*Tárcio Rodrigues Bezerra*
trb@fapeal.br

[1]*Walfredo Costa Cirne Filho*[+]
walfredo@dsc.ufpb.br

[1]*J. Antão Beltrão Moura*
antao@dsc.ufpb.br

[1]Universidade Federal da Paraíba - UFPB/Campus II
Centro de Ciências e Tecnologia - CCT
Departamento de Sistemas e Computação - DSC
Laboratório de Sistemas Distribuídos - LSD
Av. Aprígio Veloso, 882
58.109-970, Campina Grande, Paraíba
http://www.dsc.ufpb.br/~lsd

[2]Escola Técnica Federal de Alagoas - ETEFAL
Coordenadoria de Eletrônica e Processamento de Dados
Av. Barão de Atalaia, S/N, Poço
57.000-000, Maceió, Alagoas

ABSTRACT

The phases of getting a software ready and introducing it into a target-market are seldom emphasised on traditional software engineering process models. Nevertheless, these phases are of the utmost importance in the production process of many commercial applications. Within a number of activities carried out on these phases, protection of intellectual property is a key issue that must be tackled. In this paper we present the design of a license management service that can be used to avoid illegal utilisation of applications, specially those executed over a network. The unique architectural model of our license management tool confers on it important characteristics such as robustness and flexibility, not present on other solutions currently available. We compare our approach with others and show that ours is more suitable for a wider range of applications.

Keywords and phrases: license management, fault-tolerance, commercial applications' production process, software developing tools.

[+] On leave at University of California at San Diego developing a PhD program supported by CAPES (grant BEX2428/95-4)

1. Introduction

Software engineering has evolved immensely throughout the past two decades and there are a number of software developing models that have been used successfully to produce high quality reliable applications [Aoyama 93, Basili et al. 95]. However, it has also been observed that a large number of systems has not been able to attain the same level of success using these models [MMM 95].

More specifically, software engineering for most of the small and medium size commercial application projects does not seem to follow the same patterns that have been used in large ad hoc systems. The excessive emphasis that conventional software engineering models place on the development phase can be deemed as one possible cause for this. For a large portion of the software industry, successfully developing a system that matches a correct specification is only half of the story. Most of the cost of the product is not related to the development phase, but with activities related to getting the system ready and available to come onto the market. Common activities that have to be carried out in these often underestimated phases are: packaging, alpha and beta testing, distribution, technical support and marketing of the product. Only recently some models have been proposed that take these aspects into consideration [Yeh et al. 91, Potts 93, MMM 95].

Within the highly competitive market of commercial software applications, lowering development costs, enhancing product quality and customers' satisfaction, are important factors that must always be sought after by developers. If suitable models are not available, one approach that can be followed is to use appropriate developing tools [BMJH 96].

In this paper we present the design of a license management tool that can be used to avoid illegal utilisation of software applications, specially those that can be used over a network. As will become clear throughout the paper, protection against illegal use of a software product is an issue that must be tackled on most of the activities associated with its preparation and distribution phases. For instance, it is an usual practice to make non-mature versions of a software product available for prospective customers for beta testing prior to software distribution. Beta test copies are normally protected by some mechanism that makes its use impossible after a number of executions have been performed or after some expiration date has elapsed. Also, distribution costs can be substantially cut down if the protection scheme is flexible enough to allow evaluation copies to be upgraded to full-functional ones without the need for re-sending new media or supporting re-installation. Further, licensing based protection schemes can be used as an important marketing strategy when defining final product pricing policy [Élan 95].

The remaining of the paper is structured as follows. Section 2 presents the main mechanisms that have been used in order to protect software against illegal use; the principal characteristics of each approach are discussed, with their advantages and disadvantages being briefly studied. In Section 3 we introduce Bouncer, a distributed, fault-tolerant and flexible license management service that can be used to avoid illegal use of commercial applications. Section 4 brings a comparison between Bouncer and other license management tools; we emphasise the criteria we believe that indicate Bouncer to be a better solution than others currently available. Finally, Section 5 concludes the paper with some closing remarks.

2. Protecting Software against Illegal Use

Protecting intellectual property is a rather controversial issue that has been discussed for quite a long time. Legal protection such those supported by copyright and patent regulations is the main resource used by those interested in preserving their intellectual property [RS 87]. However, for the software industry, legal protection alone does not seem to have the effect its users would like to achieve. According to statistics from the Software Publishers Association (SPA), software piracy was responsible for a revenue loss to software publishers of nearly US$ 13 billion in 1993 and over US$ 15 billion in 1994 [Aladdin 97]. Only in the United States' market, it is estimated that about 33% of all software commercialised is obtained illegally. In other markets the situation is even worse. For instance, the Asian market, one of the biggest, and certainly one of the most important software markets in the world, piracy represents around 80% of everything that is sold [Élan 95].

There are basically two reasons that turn illegal appropriation of software so frequent and attractive. Firstly, since there is no loss of quality in the digital copying process, digital copies are as good as their original counterparts. Secondly, many computer users either think that software piracy is not really a crime, or believe that tracing of illegal software use is not carried out effectively (which seems to be true).

Long ago software publishers have realised that software piracy must be prevented, and a number of protection schemes have been developed. The first protection schemes produced were based either on hardware padlock devices or software activation keys. For software applications protected by a padlock to work it is necessary that a special device (the padlock) be present at the computer where the application executes. A typical example of a padlock is a device that must be attached between the serial interface and the mouse for the application to work. Protection schemes based on software keys, on the other hand, require that, when installing the product, the user types in a special sequence of characters (the activation key) supplied by the software publisher, without which the installation process cannot be successfully completed.

Although padlock based protection schemes are very secure, making virtually impossible the illegal utilisation of software products, there are many concerns regarding its use. This is mainly because of the flexibility restrictions imposed to the application users and the need to adapt padlocks to the evolution of interfaces and peripherals. In fact, nowadays very few (if any) products use this kind of software protection. Software key based schemes are not very promising either, since it is always possible for a dishonest user to get access to the activation key, and install an illegal copy of the product. Despite the relatively low level of security offered by this kind of mechanism, many products still use it, mainly because of its simple implementation and low cost.

With the rapid proliferation of local area networks (LANs) and the migration from stand-alone environments to distributed ones, the issue of software protection has gained a completely different perspective. The facility with which resources can be shared within a LAN, makes software piracy not only easy to be achieved by dishonest users, but also difficult to be avoided by honest ones. Within this framework license management services (LMS) emerged as a solution for software protection [Casey 97].

Software licenses represent the rights and the utilisation rules for a software product, as they have been agreed by a particular software publisher and a customer that acquires its product. An LMS guaranties that software utilisation complies with the agreement represented by its license. There are several attributes that can be used when defining a license agreement; the list of licensing policies that follows gives a better idea of the flexibility and power of this kind of software protection mechanism.

- **Concurrent licenses**: when applications are executed in a networked system, a number of licenses (normally smaller than the total number of seats in the network) can be acquired to allow concurrently use of the application by a limited number of users; licenses float from user to user, in a per-activation basis; in networked systems this is by far the most common licensing policy used by software developers;

- **Node-locked licenses**: licenses are available only at a single host; this is a useful policy when the developer wishes to use the license management technology to protect personal, single user systems, as opposed to networked application using floating licenses; many LMSs use themselves this kind of protection;

- **Demo licenses**: these licenses are useful when developers want to distribute fully-functional evaluation copies; applications protected by this kind of policy will either execute for a maximum number of times, or until an expiration date has elapsed;

- **Domain licenses**: this policy restricts application utilisation to a specific Internet domain for department or company-wise licensing; this can be applied by either the user or the developer to restrict use of an application outside an specific domain;

- **Reserved licenses**: reserve a number of licenses to specific groups of users; this is useful when the user wishes to guarantee a certain level of availability of the application for a group of users;

- **Shared licenses**: allow many executions of a particular set of applications to share a single license; this is normally used to allow multiple execution of an application, provided that all execute on the same host and X-display.

There are many other attributes that can be used in defining a licensing policy. Furthermore, most of the policies described above can be combined to produce even more sophisticated licensing policies.

The most common way of structuring an LMS follows the client-server model. Protected software products are "wrapped" by a client code that is responsible for contacting the license server in order to acquire an execution license. The server part of the LMS normally executes as a daemon at a particular host - the server host - within the network (in stand-alone configurations, both application an LMS execute on the same host). Its job is to receive requests from client applications and verify if their execution is in accordance with the license agreement. Based on information gathered each time a request is received and on licensing information available at the server host, the license server can respond to the client in an appropriate way. The behaviour of the client after receiving the response from the server is application dependent. Normally, if the response is positive, i.e. there is an execution license available for that client, the client will call the application which will continue its normal execution, otherwise, the client will abort the application execution with a suitable error message. In case the application succeeds in getting a license, after its execution is completed control is returned to the client, so that it can contact the server releasing the license used. Figure 1 below shows the interactions between client and server on a conventional LMS.

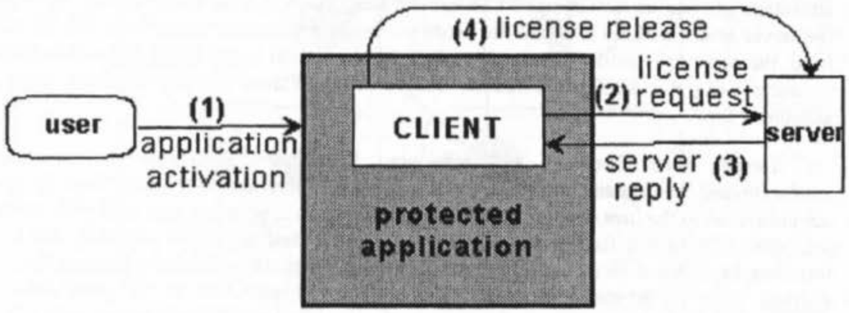

Figure 1: client and server interaction on a conventional LMS

Piracy is prevented by an LMS because the license server can only execute on a specific authorised host (i.e. it is node-locked). Further, the licensing information is encrypted in such a way that only the LMS can recognise its contents; any attempt to change this information will invalidate it.

There are two ways to tie up the license server to the server host. The first approach is to use a system specific signature (often simply the server host identification); the signature is generated by the software publisher after getting the appropriate specific information from the user and encrypting it; the resulting signature is then sent to the user, so that it can be stored in a file accessible by the host server. The second approach is to use a hardware device (a "dongle") that contains licensing information and must be connected to the server host (normally through a parallel port). In this case, it is also possible to have the license server implemented inside this hardware device.

The procedures followed by software developers in order to integrate an LMS into their software products vary slightly, depending whether the license server is tied up with the server host through a software signature or a hardware device. However, in both cases developers have to add a few subroutine calls into their application so that it can contact the server trying to acquire a license for execution. Also, appropriate testing must be conducted, the need for changes in the documentation must be assessed, and a number of cross departmental business decisions must be made (e.g. definition of new pricing strategies, provision of technical support for the LMS, etc.).

Additionally, if software signature is used, a license management group must be set up. Creation and recording of licenses are the main activities carried out by this group. Obviously, access to the license generation procedure should be restricted to as few individuals as possible within the corporation. On the other hand, in order to maintain customers' satisfaction high, the license management group has to respond quickly to all customers' requests. These two requirements are somehow conflicting and possibly difficult to be attained.

When hardware devices are used, they have to be enclosed with the shrink wrapped software (one device per license server). Devices can be supplied with pre-programmed license information, obviating the need for any interaction between software publishers and customers, and consequently the maintenance of a license management group. Portability is another advantage of tying up license servers and server hosts through a hardware device. To change the server host it suffices to install the hardware device on the new server host. On the other hand, the cost of the extra device and the limitations imposed to the distribution process (e.g. products using this kind of protection cannot be distributed via the Internet), must be accounted for.

Despite their flexibility and efficiency, LMSs are susceptible to operational malfunctioning, contributing in some cases to a substantial increase on the work load of system administrators at the user environment. In order to make an application protected with an LMS available to its users, the system administrator has to deal with new activities related to installing hardware devices, starting daemons at specific hosts, maintaining license files, etc. Further, since robustness is a fundamental property of an LMS, if the LMS does not incorporate suitable fault tolerance features, its the system administrator's task to maintain the license server operational (e.g. re-starting crashed servers, releasing licenses held by applications that had exited abnormally, etc.).

Although LMSs constitute a very promising solution for the problem of avoiding software piracy, the discussion above has pointed out a number of drawbacks presented by them. In the following section we introduce an LMS that tries to minimise most of the inconveniences common on other LMS reported in the literature. Later, in Section 4, we return to the issue of analysing advantages and disadvantages of LMS tools and show how our approach differentiates from the ones currently available.

3. The Bouncer Model for Software Protection

3.1. Service Architecture

Unlike most LMSs that follow a client-server programming model, the Bouncer LMS adopts a hybrid model of distributed programming, merging the client-server model with the peer-group model. In the context of a single host, the Bouncer LMS uses a client-server model similar to the model used by conventional LMSs discussed previously. In order to acquire an execution license, the client requests it to a server that is always executing on the same host the client is executing (if no such server is available the client itself is responsible for activating the local license server). Requests to release execution licenses are also sent to this local server. Differently from other approaches, the license server is a distributed server that executes on all hosts that hold an active copy of a protected application. The distributed server programming model follows a peer-group approach. In this way, the Bouncer LMS takes advantage of the simplicity provided by a client-server model, and the potential greater robustness of a peer-group model. Figure 2 shows the interactions between clients and servers on a Bouncer LMS.

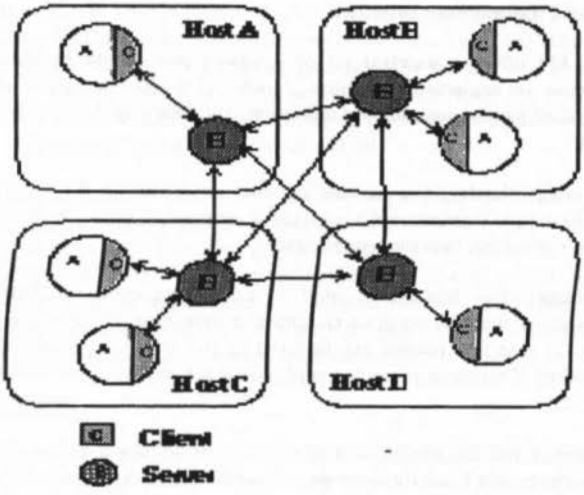

Figure 2: hybrid programming model of the Bouncer LMS

Again, the client part of the LMS is introduced into the application. When trying to acquire an execution license, the client sends a request to the Bouncer server executing on the local host. Any instance of the license server upon receiving a local license request, atomically broadcasts this request to all other instances of the Bouncer server executing on the network. By collecting and processing these broadcasts, which are guarantied to be received by all functioning servers in the same order, every instance of the license server is able to maintain a consistent global knowledge of the licenses that have been granted, and therefore can decide whether a local request should be granted or denied. Before exiting, an application that holds an execution license must return it through a call to the local server; this in turn, atomically broadcasts the request to the other servers. A more detailed description of the protocol executed by clients and servers implementing a Bouncer LMS can be found in [BBC 97].

Several licensing policies can be implemented depending on the way servers are tied up to hosts. In its simplest protection form no ties are enforced. Applications and servers can execute on any host in the network, however only a limited number of applications can execute concurrently. Servers executing on every host with an active application are respônsible for keeping track of the number of applications concurrently executing at all times. New execution licenses are granted only if the maximum number of concurrent executions has not yet been reached. On the other spectrum an application can define sophisticated licensing policies which restrict execution of applications to a collection of hosts within the network. Licensing policies are specified through a software signature stored on a file (or files) accessible by all hosts on which protected applications may execute. Signatures, among other things, can be used to specify the set of hosts or the domain where servers may execute. It is worth noting that in the simpler case, neither hardware devices, nor software signatures are needed; as we discuss in Section 4, this can be a very attractive feature for an LMS, despite its limited protection capability.

3.2. Application Programming Interface

Bouncer's API offers a standard set of functions that should be used by software developers in order to implement the client portion of a Bouncer LMS and have their applications protected against unauthorised utilisation. The following functions are provided by the Bouncer API:

- **B_Request()**: this function requests an execution license to the local license server; the application is blocked until a response is returned; if no server is executing first a server is started and then the request is sent;
- **B_Release()**: this function is used to release an execution license; when the application is about to finish its execution, it must return its execution license to the server, so that this license can be used in the future by other activation of the application; if no more protected applications are executing on that host, the server also exits;
- **B_Check()**: this function is used by applications holding a license, in order to verify the existence of a local Bouncer server (see Section 3.3 below); if there is no server executing, a new server is re-started and a license re-validation request is issued;
- **B_Monit()**: this function requests information about applications being protected by the license server (e.g. host where they execute, number of licenses used, number of licenses available, etc.); it is used to implement license metering tools [Casey 97].

Alternatively, it is possible for software developers (or even publishers, in this case) to use a wrapper to make the software protection scheme transparent to the application. When the user executes the application, first the wrapper (playing the role of the client) is executed and requests an execution license to the local server. Depending on the response received, the actual application is activated or the wrapper aborts its execution. After the execution of an application the wrapper sends a request to release the license just used.

3.3 Fault Tolerance Properties

As mentioned before, the functioning of an LMS may be jeopardise by the occurrence of faults. Faults, in turn, may result from both intentional (e.g. attempts to break software protection) and non-intentional (e.g. a host crash) events. In any case, an LMS must continue providing its service despite the occurrence of faults, i.e. an LMS must incorporate provisions to tolerate as many faults as possible. There are basically four faulty behaviours that can be observed during the execution of a Bouncer LMS, and should be tackled: i) a (local) server has failed; ii) a false server tries to impersonate a Bouncer server; iii) the application has failed before releasing its license, generating orphan licenses; iv) the network is partitioned. We now discuss how each of these faulty behaviours are dealt with by the Bouncer LMS.

Server Failure

This can happen due to crashes on either the license server or the server host. If the license server has crashed, the client part of the applications executing at that host will automatically re-start the server. This is achieved by introducing into the application code frequent calls to the B_Check() function. A re-started server receives information about the current allocation of licenses from other servers in the network (if the crashed server was the only server available, the re-validation requests issued by the B_Check() calls will suffice to update its state). On the other hand, if the host has crashed, since the local applications have also crashed, there is no need to re-start the server (eventually a protected application will be activated on that host and a new server will be started).

The group of local servers that collectively implements the license server executes a membership control protocol whose main objective is to atomically detect server failures [BBC 97]. Thus, when the server fails, two possibilities may arise. In the first situation, all other functioning servers will detect that a server has failed. The only action that they take is to release any license held by applications executing at the host where the faulty server was executing. In the second case, the server recovery is quick enough to prevent the membership protocol from detecting its failure, therefore no further action is needed.

Server Impersonation

A dishonest user might try to break the Bouncer protection by installing a fake license server which always grants execution licenses to applications. To eliminate this problem, all messages exchanged by client and servers are signed at their origin and authenticated at the destiny. There are well known cryptography techniques that can be used to implement such service [Schneier 96].

Client Failure

Unlike server failures, treatment of client failures are normally postponed until the point where a license request may be denied (e.g. because the maximum number of concurrent licenses granted, for a particular application, has been reached). At this point every server initiates a local search for orphan licenses. For every local application that has crashed, the local server broadcasts a license release request to the other servers. In this way, every server can consistently process all release requests issued by the several orphan detection procedures performed, and maintain its consistent view of the licenses in use.

Network Failure

Network partitioning is a difficult problem to solve. In conventional LMSs based on a centralised license server, the partition of the network causes the unavailability of the service to applications that were executing (or would be executed) at the partitions where the server host is not present, until the network is fixed. In the case of a distributed LMS, the partition of the network might permit the replication of the number licenses in use, allowing temporary illegal activation of an application. However, once the network is re-unified, the correct licenses is enforced, and illegal activation of applications is detected and, depending on the developer's decision, aborted.

4. A Comparison between Several License Management Tools

It has long been recognised that the utilisation of appropriate tools is an effective way to improve productivity and product quality in the software production process. However, introducing a new tool incurs costs that should not be overlooked [BMJH 96]. Besides the acquisition cost, training project members to use the tool and converting or adapting current designs and production processes to incorporate the new technology are other sources of expenditure that must be accounted for. Furthermore, in the case of software protection tools, the impact that these tools might have at the users' environment is an important matter to be pondered.

In order to assess the quality and suitability of a tool, software publishers normally follow well defined evaluation guidelines. For example, the ISO 9126 standard proposes a set of characteristics such as functionality and usability, among others, that may be used in estimating the quality of a software product [ISO9126]. We will follow the criteria listed below, to develop our comparison between the Bouncer LMS and other LMSs available.

- **Functionality**: to evaluate how well the tool performs its task;
- **Usability**: to evaluate how easily the tool can be used;
- **Reliability**: to evaluate the robustness of the tool; and
- **Flexibility**: to evaluate how much effort is needed to integrate the tool into the production process.

As far as we know, all LMSs currently available adopt the client-server model discussed in Section 2. However, they can be divided into two different classes, depending on the method used to node-lock servers to specific hosts. Thus, there are those which use software signatures, commonly found protecting applications executing on workstation-based systems (e.g. FlexLM [FlexLM 96, FlexTO 96], ÉlanLM [ÉlanLM 95], iFOR/LS [Gradient 95]), and those normally used to protect applications executing on PC-based systems, which adopt hardware devices (e.g. HASP [Alladin 97], SentinelLM [Rainbow 97]). In the discussion that follows we use the term software LMSs to refer to LMSs using software signature schemes, and hardware LMS to refer to LMSs using the hardware device approach.

Functionality

The main functionality of an LMS is to protect software against illegal utilisation. Thus, its functionality can be measured by assessing how secure is the method used to implement their protection scheme. There are two issues to handle: i) preventing illegal copying of the protected application; and ii) preventing illegal utilisation of the protected application. The first problem is normally solved by tying up the execution of servers to specific and well defined hosts, whilst the second problem is solved by storing encrypted licensing information at the user's environment in such a way that only the LMS servers may recognise it; further, any attempt to modify the information must be detected by the server. Since cryptography is the heart of the security mechanisms used by all LMSs, they use cryptography algorithms that are virtually impossible to break [Schneier 96].

In software LMSs, like Bouncer, the information describing which host (or hosts) may execute the server is included into the signature file, which also contains licensing information. The signature file is encrypted by the software publisher and sent to the user to be stored on a file accessible by all LMS servers. Messages exchanged between the protected application and the server are also encrypted. In hardware LMSs, the location of the "dongle" itself defines in a straightforward way the host where the server executes. Software publishers generate encrypted licensing information which is stored into the hardware device. Once the device is installed at the user's environment, the information can be accessed by the LMS server. Protected applications issue calls directly to the hardware device, which encrypts a reply and sends it to the application, therefore, it is not possible to use another device to impersonate the original one. In both cases, any attempt to change the licensing information is detected by the server.

Apart from using cryptography on signature files and "dongles" and on the information exchanged by clients and servers, a number of mechanisms are often incorporate to LMSs, in order to discourage the action of hackers trying to reverse engineer the protection scheme. Common mechanisms to achieve this goal are the disabling of interrupts used by simple debuggers to prevent hand-debugging and the inclusion of 'spaghetti' code (e.g. using computed jumps or self-modified code) to prevent hackers from easily walking the security code.

Using the mechanisms described above, the Bouncer LMS as well as all LMSs currently available, if properly used by software publishers, can provide very good software protection [NSTL 95].

Usability

Nearly all LMSs available offer a similar API and provision for the utilisation of a wrapper to make the incorporation of the protection mechanism easier and programming language independent. When the wrapper is not used, applications must incorporate calls to the functions provided by the LMS API. Normally, when the application first starts it calls a function to acquire an execution license, if the license is successfully acquired, the application main body is executed. Just before exiting, the application calls a function to release the license it holds. Additionally, it may be required that the application frequently calls a function that checks if the license held is still valid (see the reliability issues below). This is commonly achieved by starting a simple watchdog thread. We can conclude, then, that all LMSs, including Bouncer, are fairly simple to be used by software developers.

Reliability

All LMSs based on the client-server model suffer, in greater or lesser extent, from reliability problems. First, when the server is down, applications cannot be used. Also, all applications which were executing whilst the server had gone down are normally aborted or blocked.

Servers can fail for a variety of reasons - the server host has gone down, the server daemon has been inadvertently killed, etc. One approach often used to tolerate such failures is to replicate the license server on hosts which fail independently. All software LMSs available provide mechanisms for allowing server replication on pre-defined hosts. Some LMSs (for example, iFOR/LS) implement a simpler replication strategy whereby servers divide the total number of concurrent licenses among them, i.e. server failures produce a gentle degradation of the service. However, in all cases, the system administrator at the user environment is responsible for detecting the failure of servers and re-starting faulty servers to re-establish system dependability level. Further, if all hosts pre-defined to execute license servers are unavailable (e.g. for maintenance), the service cannot be re-started until at least one of these hosts is made available or a new license file allowing new hosts to execute the license server, has been received from the software publisher.

Even hardware LMSs with the server functionality hardwired into the "dongle" device can fail, since the host where the device is installed can fail or the device can stop functioning. In the former case the system administrator has to detect the host failure and install the device on another functioning host. In the latter case, recovery can only be attained by contacting the technical support staff of the software publisher and ordering a new operational hardware device. Although the reliability of the hardware device is high enough to minimise the need for replacement due to malfunctioning, the usual practice of moving the hardware device from one machine to another within the corporation may increase the number of defects and losses of the hardware device.

Unlike all other LMSs, the Bouncer LMS allows server recovery without the intervention of the system administrator. This is achieved by adding to the application code the necessary server start up procedures as well as frequent calls to check the existence of the server. When a server failure is detected, a new server is automatically re-started. If necessary, the new server gets its status from a server running on another host. Also, the peer-group architecture of Bouncer provides transparent replication of the service, producing a highly available LMS.

A second problem that may occur is client failure. When the application fails before releasing the execution license it holds, that license is lost until recovery is carried out. All LMS available, but the ÉlanLM[1], need intervention from the system administrator for recovering orphan licenses. As presented in Section 3.3, the Bouncer LMS recovers transparently from all protected application failures. Before denying an execution license the servers execute local procedures that try to detect orphan licenses generated by local applications that have crashed. Once these licenses are detected, the server issues the corresponding release calls, freeing them.

Finally, network failures are also a source of problems. In all LMSs available, network partition leads to service unavailability for all applications executing (or to be executed) at a particular partition where there is no license server executing. Also, the behaviour of replicated servers that have been disconnected may cause additional problems. In all cases, the network partition has to be detected and fixed by the system administrator. In most systems, once the partitioning is fixed, the service is operational again. However, when there are replicated servers executing, sometimes the system administrator has to re-start all servers and any surviving application.

As discussed in Section 3.3, the Bouncer LMS deals with network partition in such a way that it tries not to penalise users or system administrators. Even when the network is broken, it is possible that applications in any of the resulting partitions may execute. The only problem is that users may temporarily (i.e. during the time the network is broken) execute applications illegally, since each partition will resemble the original network, resulting in a replication of the licensing agreement. However, once the system administrator has detected

[1] It is not clear how the ÉlanLM LMS deals with the recovery of orphan licenses, but its documentation [ÉlanLM 95] argues that the product can tolerate this kind of failure transparently.

and fixed the network, the functionality of the protection mechanism is restored. Network partitioning normally affects a large number of applications in a networked system, therefore, system administrators are already supposed to monitor and fix this kind of failure. Further, since network partitioning disturbs many applications, it is unlikely that a dishonest system administrator would deliberately break the network in order to execute protected applications illegally.

Flexibility

Besides including the necessary API calls into the protected application, integrating an LMS to an application requires a number of activities that must be conducted by software developers or publishers. First, documentation must be changed in order to incorporate information on how to install servers and have them automatically started at boot time, or how to install hardware "dongles", and how to deal with LMS failures. Second, if a software LMS is used, a license management group must be set up. Third, if a hardware LMS is used, the packaging phase procedures must be changed to allow the inclusion of the hardware device along with the protected software. Finally, technical support staff must be trained to deal with the new problems that the LMS may cause.

Unlike all other software LMSs available, the Bouncer LMS can provide a protection service that does not require setting up a license management group. This is because concurrent licenses, the most commonly used licensing policy, can be implemented by the Bouncer LMS without the need for a software signature. The number of concurrent executions of an application can be hardwired into the server code that accompanies the application, obviating the software signature, and consequently the necessity of a license management group. Obviously, since servers are not node-locked in any way, this protection mechanism is note as secure as one based on hardware keys or software signatures. However, for small software-houses, the extra cost of adding a hardware device for each copy sold or maintaining a license management group to deal with the generation of software signature is too high. Therefore, a less secure but cheaper solution might be a more suitable option.

As mentioned before, Bouncer's peer-group model eliminates the need for starting servers and monitoring client and server failures. As a consequence, few changes (if any) will have to be made on the documentation of protected applications, and less burden is put on system administrators. In its most secure form, Bouncer's only requirement is that a file containing the software signature be accessible by all hosts where the protected application may execute. An appropriate installation script may also hide this fact from the user. Further, since there is no need for a hardware device to be shrink wrapped with the software, applications protected by the Bouncer LMS may use electronic distribution procedures.

In summary, the Bouncer LMS provides a very flexible protection service, with varying security properties and associated costs. Software developers may choose from a simple protection scheme allowing only license agreement enforcement with little modifications on the software production process, to sophisticated ones allowing both license agreement enforcement and copying prevention, but requiring some modifications on the production process. In all cases however, little extra technical support is needed, since the protection service is transparent to users and system administrators.

5. Conclusions

Software production can be a very costly activity, demanding large investments of money and time. Firstly, there is the need for contracting qualified personnel (programmers, software engineers, managers, technical support staff, etc.); secondly, several software development tools (compilers, third party libraries, CASE tools, etc.) and different hardware platforms (e.g. for porting) must be acquired; thirdly, marketing the product accounts for costs related to hiring graphical designers and marketing experts as well as producing and publishing publicity material; finally technical support has to be provided. Nevertheless, the product of all this effort can be illegally appropriated though a non-authorised copy.

In this paper we have discussed the role of software protection within the commercial software production process. We have presented several ways of protecting software, paying special attention to protection mechanisms based on license management services (LMSs). We have analysed their advantages and drawbacks, and introduced an LMS that tries to eliminate problems founded on others approaches currently available.

Although our approach presents robustness and flexibility characteristics not present in other solutions, it also presents some drawbacks. First, support-free protection solutions can only be provided with substantial decrease on the security of the protection mechanism. Second, replicating the server on every host that executes a protected application increases the amount of resources needed to execute the application. Finally, illegal executions of a protected software are possible when the network is partitioned. Despite these disadvantages, we believe that the advantages presented by our approach outweigh the disadvantages, making it a better solution for a wider range of applications.

Currently an implementation of the Bouncer LMS is under development. The enterprise receives financial support from the Brazilian program Softex 2000, through one of its Genesis consortium [Poligene 97].

Acknowledgements

Authors would like to thank financial support provided by CNPq (grants 380.076/94-2, 300.646/96-8), CAPES (grant BEX2428/95-4), and CNPq/PNUD (grant 680071/94).

References

[Alladin 97] Alladin Knowledge Systems, Inc. HASP Home Page, http://www.aks.com/, 1997.

[Aoyama 93] M. Aoyama, "Concurrent Development Process Model," IEEE Software, Vol. 10, N. 4, pp. 46-55, July 1993.

[Basili et al. 91] V. Basili et al., "SEL's Software Process-Improvement Program," IEEE Software, Vol. 12, N. 6, pp 83-87, November 1995.

[BBC 97] T.R. Bezerra, F.V. Brasileiro, and W.C. Cirne Filho, "Bouncer - Um Serviço Distribuído e Tolerante a Faltas para Controle de Licenças de Software," (in Portuguese), submitted to the VII Symposium on Fault-Tolerant Computers, March 1997.

[BMJH 97]	T. Bruckhaus, N.H. Madhavji, I. Janssen, and J. Henshaw, "The Impact of Tools on Software Productivity," IEEE Software, Vol. 13, N. 5, pp. 29-38, September 1996.
[Casey 97]	L.. Casey, "Network License management Solutions," Rainbow Technologies, http://www.rnbo.com/SENTINELLM/Article.html, 1997.
[Élan 95]	Élan Computer Group. Executive Brief of License Management, http://www.elan.com/ebintro.html, 1995.
[ÉlanLM 95]	Élan Computer Group. Élan License Manager Technical Overview, http://www.elan.com/elanlm.html, 1995.
[FlexLM 96]	GLOBEtrotter Software, Inc. FlexLM End User Manual, http://www.globes.com/manual.html, 1996.
[FlexTO 96]	GLOBEtrotter Software, Inc. FlexLM Technical Overview, http://www.globes.com/flexto.html, 1996.
[Gradient 95]	Gradient Technologies, Inc., iFOR/LS Quick Start Guide, Version 2, http://www.gradient.com/, 1996.
[ISO9126]	Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for their Use, ISO-9126, 1991.
[MMM 95]	L.M.F. Martins, J.A.B. Moura, and A.F.C. Medeiros, "R-Cycle: Um Molde para o Processo de Produção, Disponibilização e Evoluçaão de Software," (in Portuguese), Proceedings of the IX Brazilian Symposium on Software Engineering, 1995.
[NSTL 95]	National Software Testing Laboratories. NSTL Dongle Security Comparative Evaluation, October 1995.
[Poligene 97]	Núcleo Poligene. Home Page, http://www.dsc.ufpb.br/~genesis.
[Potts 93]	C. Potts, "Software Engineering Research Revisited," IEEE Software, Vol. 10, N. 5, pp. 19-28, September 1993.
[Rainbow 97]	Rainbow Technologies, SentinelLM Home Page, http://www.rnbo.com/SENTINELLM/home.html, 1997.
[RS 87]	D. Remer and E. Stephen, Legal Care for your Software - a Step by Step Guide for Computer Software Rights and Publishers, 3rd edition, 1987.
[Schneier 96]	B. Schneier, Applied Cryptography, 2nd edition, John Wiley & Sons, Inc., New York, 1996.
[Yeh et al. 91]	R.T. Yeh et al., "A Commonsense Management Model," IEEE Software, Vol. 8, N. 6, pp. 23-33, November 1991.