Tárcio Rodrigues Bezerra

Bouncer - Uma Solução Distribuída para Controle de Licenças de Software

Dissertação submetida ao corpo docente da Coordenação de Pós-Graduação em Informática da Universidade Federal da Paraíba - Campus II, como parte dos requisitos necessários para a obtenção do grau de Mestre em Informática.

Área de concentração: Redes de Computadores e Sistemas Distribuídos

Francisco Vilar Brasileiro - UFPB - CCT - DSC
Orientador

Walfrêdo da Costa Cirne Filho - UFPB - CCT - DSC
Co-orientador

Campina Grande, Paraíba, Brasil

BOUNCER - UMA SOLUÇÃO DISTRIBUÍDA PARA CONTROLE DE LICENÇAS DE SOFTWARE

TÁRCIO RODRIGUES BEZERRA

DISSERTAÇÃO APROVADA EM 30.12.96

PROF. FRANCISCO VILAR BRASILEIRO, Ph.D.
Presidente

PROF. MARCELO ALVES DE BARROS, Dr.
Examinador

PROF. ALCIDES CASALVARA, Ph.D
Examinador

CAMPINA GRANDE - PB

A Deus, por tudo.

A mim, pela determinação.

A minha família, com carinho.

DIGITALIZAÇÃO:

SISTEMOTECA - UFCG

// Agradecimentos

Aos meus orientadores, Prof. Francisco Brasileiro - o Fubica, Prof. Walfrêdo Cirne Filho - o Banal e Prof. Jacques P. Sauvé, por partilharem comigo um pouco dos seus extensos conhecimentos; pelas valiosas discussões, pela paciência e amizade, indispensáveis para a conclusão deste trabalho.

A Gedeon J. Santos (grande *stress-man*), pois este trabalho foi fruto de suas idéias.

Aos Professores Stênio Flávio L. Fernandes e Guilherme Ataíde Dias, da ETFAL, pelo grande apoio, principalmente nas horas em que precisei me ausentar de minhas atividades para me dedicar a este trabalho.

Aos professores que fazem a Coordenadoria de Eletrônica e Processamento de Dados da ETFAL, pelo incentivo.

A toda a galera do desenvolvimento da Light-Infocon, pela amizade, compartilhamento de conhecimentos técnicos e pelos momentos de descontração (biritas, guerras de bolinhas de papel, o monstro do bat gut, o observador, etc.).

A todos os bons amigos que fiz no Mestrado em Informática da UFPB, bem como à especial atenção dispensada por Aninha, da COPIN, sempre empenhada em me ajudar quando solicitada.

A Susan Miller, da GLOBEtrotter Inc., pela prestatividade em me fornecer importantes informações.

A todos os meus amigos, por acreditarem na minha capacidade.

À música, por ser meu prazer nas horas de descontração e minha grande válvula de escape nos momentos de maior tensão.

Resumo

A pirataria é um dos mais sérios problemas que atingem a indústria de software. A possibilidade de se gerar uma cópia digital tão boa quanto sua versão original estimula o uso ilegal de produtos de software. Dentre as soluções para este problema, vem se destacando e ganhando importância o serviço de licenciamento.

O serviço de licenciamento verifica se uma determinada aplicação é licenciada (i.e., não é pirata) no momento de sua ativação. Entretanto, a inclusão de tal serviço em um ambiente computacional gera atividades adicionais de suporte técnico. Além do mais, a utilização de gerenciamento de licenças geralmente irá demandar por parte do fabricante de software a manutenção de uma estrutura de suporte técnico dedicada a atividades relacionadas com configuração de políticas de licenciamento (por exemplo, geração de chaves de software baseadas em informações do ambiente computacional do comprador para habilitar políticas de licenciamento). Isto não é adequado para pequenas empresas produtoras de software de prateleira, que precisam ter um baixo custo operacional para que seus produtos possam ter preços mais competitivos neste mercado.

Neste trabalho, apresentamos uma solução de licenciamento tolerante a faltas, auto-configurável e livre de suporte, denominada Bouncer. Estas características fazem do Bouncer uma opção de licenciamento mais adequada para o mercado de software de prateleira.

Abstract

Piracy is one of the biggest problems that the software industry faces nowadays. One fact that turns the illegal appropriation of software so frequent and attractive is that there is no loss of quality in the digital copy process. Digital copies are as good as their original counterparts.

License management services are getting increasingly more importance among the solutions to this problem. These services verify if a given application is in accordance with the revenue agreements (i.e., it is not an illegal copy). However, the inclusion of license management in a computing environment creates a number of additional technical support tasks. Moreover, license managers demand the maintenance of technical support staff by the developers'·companies to interact with customers, and executing tasks such as generating software keys based on customers' environment information and re-configuring license policies. This is not suitable for small companies structures that produces shrink-wrap software packages.

In this work, we present Bouncer, a fault-tolerant, self-configurable and suport-free licensing solution. These characteristics make Bouncer a feasible candidate for the shrink-wrap software packages licensing market.

Sumário

1. Introdução **1**

1.1. Coibindo o Uso Ilegal de Software 2

1.1.1. Proteção Legal 2

1.1.2. Proteção por Hardware 3

1.1.3. Proteção por Software 4

1.1.3.1. Proteção em Ambiente Centralizado 4

1.1.3.2. Proteção em Ambiente de Rede 6

1.2. Objetivo e Organização do Trabalho 6

2. Serviços de Licenciamento **9**

2.1. Definindo Políticas de Licenciamento 12

2.2. Principais Licenciadores Existentes no Mercado 13

2.2.1. FlexLM 13

2.2.2. ÉlanLM 17

2.2.3. iFOR/LS 18

2.3. Problemas do Serviço de Licenciamento 21

3. O Bouncer **25**

3.1. Arquitetura e Serviços 27

3.2. Interface de Programação 29

3.2.1. B_Request 29

3.2.2. B_Release 32

3.2 .3. B_Monit 32

3.2.4. B_Check 36

3.3. O protocolo Bouncer 37

3.3.1. O Protocolo Bouncer em um Ambiente Livre de Faltas 37

3.3.2. O Protocolo Bouncer em um Ambiente Sujeito a Faltas 40

3.3.2.1. Validação dos Pressupostos 40

3.3.2.2. Análise dos Possíveis Problemas e Soluções 42

3.3.2.3. O Protocolo Bouncer Tolerante a Faltas 43

3.3.3. O Processo de Auditoria de Licenças Órfãs 46

4. Infra-estrutura de Comunicação do Bouncer **49**

4.1. Comunicação entre a Aplicação e o Servidor Bouncer 49

4.1.1. O Modelo Cliente-Servidor 49
4.1.1.1. Endereçamento 50
4.1.1.2. Primitivas Bloqueantes x Primitivas não Bloqueantes 51
4.1.1.3. Primitivas Buferizadas x Primitivas não Buferizadas 52
4.1.1.4. Primitivas Confiáveis x Primitivas não Confiáveis 53
4.1.2. Remote Procedure Call 54
4.1.2.1. Princípios Básicos de Operação do RPC 54
4.1.2.2. Passagem de Parâmetros 56
4.1.2.3. *Binding* Dinâmico 58
4.1.2.4. Semântica do RPC na Presença de Falhas 59
4.1.3. A Utilização de RPC no Bouncer 62
4.2. Comunicação entre os Servidores Bouncer 63
4.2.1. Comunicação em Grupo 64
4.2.1.1. Grupos Fechados x Grupos Abertos 64
4.2.1.2. Controle de Pertinência de Membros 65
4.2.1.3. Endereçamento 66
4.2.1.4. Atomicidade 66
4.2.1.5. Ordem na Entrega das Mensagens 66
4.2.2. A Utilização de Comunicação em Grupo no Bouncer 67
5. Conclusão **72**
5.1. Contribuições do Trabalho 74
5.2. Sugestões para Trabalhos Futuros 75
Bibliografia **78**
Apêndice **81**

Lista de Tabelas

2.1 Vantagens do uso de gerenciadores de licença. 12

2.2 Funções disponíveis na API do FlexLM. 16

2.3 Funções disponíveis na API do ÉlanLM. 18

2.3 Funções disponíveis na API do iFOR/LS. 20

2.4 Principais problemas de suporte encontrados nos gerenciadores de licença. 23

3.1 Funções disponíveis na API do Bouncer. 29

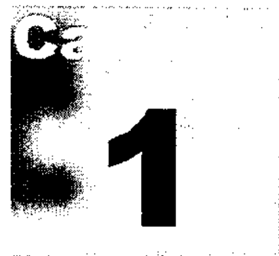
3.2 Alguns códigos de erro retornados pela API do Bouncer. 31

5.1 Quadro comparativo entre o Bouncer e outros gerenciadores de licença sobre problemas de suporte. 74

5.2 Quadro comparativo entre o Bouncer modificado e outros gerenciadores de licença sobre problemas de suporte. 77

Lista de Figuras

2.1 Uso de aplicativos integrados com um serviço de licenciamento 9
3.1 Modelo híbrido do serviço Bouncer 28
3.2 O protocolo Bouncer. 39
3.3 O protocolo Bouncer tolerante a faltas. 44
4.1 Seqüência de ativação de procedimentos remotos 55
4.2 Formação de sub-grupos Bouncer causada por segmentação da rede 69



1 Introdução

O crescimento do uso da informática está transformando a indústria mundial de software em uma das fatias de mercado que mais volume de recursos movimenta. Muito é investido em pesquisa e desenvolvimento de novos produtos e tecnologias.

Software é um artigo que pode custar milhões de dólares para ser desenvolvido. Estão envolvidos neste processo custos como salários de profissionais altamente especializados (analistas e programadores), ferramentas de desenvolvimento (compiladores, bibliotecas), geralmente dispendiosas, aquisição de diversas plataformas de hardware para porte destes produtos, entre outros custos. Os gastos não param por aí. Para que se atinja a fase de comercialização de um software, o envolvimento de profissionais como *designers* gráficos, analistas de mercado para trabalharem no *marketing* do produto, estudando características negativas dos produtos concorrentes e como estas poderão ser utilizadas para influenciar seus potenciais compradores, divulgação em mídia especializada, além da montagem e manutenção de uma infra-estrutura para suporte técnico ao produto são etapas tão cruciais quanto a própria codificação, pesando consideravelmente no orçamento do desenvolvedor. Entretanto, o produto de todo este árduo processo pode ser indevidamente apropriado através da geração de uma cópia digital não autorizada (pirata) sem qualquer tipo de ônus.

Segundo estimativas da *Software Publishers Association* (SPA), o índice de pirataria no mercado americano é de cerca de 33%, ou seja, um terço de todos os produtos de software dos Estados Unidos é obtido ilegalmente [Élan 95]. Este percentual é ainda maior no mercado mundial. Ainda segundo a SPA, US$ 12 bilhões por ano deixam de ser captados pela indústria de software por causa da pirataria.

A pirataria no mercado de software para PCs chega a representar quase que 50% do volume de recursos movimentado por este segmento, sendo cerca de US$

8 bilhões captados através da venda legal de software e US$ 7.5 bilhões movimentados pelo mercado negro. Na Ásia, um dos maiores e, certamente um dos mais importantes mercados de software do mundo, os prejuízos com a utilização ilegal de software representam 80% de tudo o que foi vendido em 1993. Foram vendidos US$ 1 bilhão em software legalizado contra US$ 4 bilhões captados através da venda de produtos pirateados. As projeções para o ano de 1997 são de que esta relação tende a se manter (US$ 4 bilhões em software legal contra US$ 13 bilhões em software ilegal, ou seja, 76% do software vendido é ilegal).

Um fato que torna a apropriação indevida tão popular é que, ao contrário de cópias analógicas, a cópia digital mantém a mesma qualidade de sua matriz. Por exemplo, quando uma pessoa deseja ter em sua residência um vídeo do qual gosta muito e decide fazer uma cópia pirata, faz questão de que a matriz de sua cópia seja a fita original, ou, pelo menos, o mais próximo possível desta, visando desfrutar da melhor qualidade possível de imagem e som. Esta pessoa, portanto, pensa duas vezes antes de possuir uma cópia de qualidade duvidosa de um vídeo, caso tenha condições de comprar sua versão original. Este problema não é enfrentado pelos piratas de software, que continuam desfrutando da mesma qualidade apresentada pela versão original do produto, independente de que nível de cópia esteja em seu poder (filha, neta, bisneta ... da versão original).

O maior inconveniente da cópia pirata é a falta de suporte técnico do fabricante, que, para muitos, não é um benefício tão significante que justifique desembolsar dinheiro na aquisição de um produto original.

Diante destes fatos, visando proteger seu investimento e sua propriedade intelectual, os desenvolvedores de software passaram a tomar atitudes para evitar a prática da pirataria.

1.1 Coibindo o Uso Ilegal de Software

Para reprimir o uso ilegal de seus produtos, os desenvolvedores aderiram a várias soluções. Dentre elas, podemos destacar: proteção legal, proteção por hardware e proteção por software. Nesta seção discutiremos cada uma destas alternativas, levantando suas vantagens e desvantagens.

1.1.1 Proteção Legal

Este mecanismo de proteção baseia-se na adoção de sanções ao usuário de uma cópia não autorizada de algum produto, como, por exemplo, um processo judicial. Existem várias publicações que tratam sobre proteção da propriedade intelectual e dos direitos autorais direcionados especificamente à indústria de software, trazendo, inclusive considerações sobre firmamento de contratos de distribuição, compra e venda, *royalties*, etc. [Remer 87]. Esta abordagem não gera problemas de ordem técnica. Entretanto, mostra-se ineficaz no combate ao uso

ilegal dos produtos, devido a crença na falta de fiscalização e conseqüente impunidade dos infratores.

Além disso, em algumas situações, mesmo quando existe um desejo de cumprir o contrato de compra/licenciamento do software, isto pode não ser possível, mediante, por exemplo, a ação de usuários que podem gerar cópias deste software de maneira furtiva e disseminá-la indiscriminadamente. A proteção legal necessita da alocação de tempo do administrador do sistema do cliente para monitorar o uso do produto, garantindo o cumprimento dos termos de utilização do software.

À medida em que os ambientes computacionais crescem, o policiamento dos usuários torna-se inviável para o administrador, havendo a necessidade de ferramentas que o auxiliem a manter controle sobre o cumprimento dos termos de uso de um software. Além do mais, nada garante que o próprio administrador não tente promover a pirataria em seu ambiente, fato este que tem se tornado comum e inviabilizado o uso da proteção legal de forma isolada.

Concluímos, portanto, que a proteção legal sempre é aplicável, porém, em conjunto com outras modalidades de proteção, já que esta se torna ineficaz sendo aplicada de forma isolada. De maneira geral, a atribuição de fiscal na proteção legal é dada aos administradores de sistema/redes dos ambientes computacionais localizados nas dependências dos compradores de software. Porém, o crescimento das corporações tem dificultado a ação eficiente dos administradores, que passaram a necessitar de ferramentas de auxílio ao policiamento de seus usuários.

1.1.2 Proteção por Hardware

Este tipo de proteção consiste no uso de pequenos dispositivos de hardware, vulgarmente conhecidos como **cadeados**. Como exemplos de cadeados, podemos citar: 1) conectores colocados entre o mouse e a interface serial de um PC; 2) disquete perfurado em determinada posição de sua superfície magnética, devendo este estar sempre inserido na unidade de disco flexível durante a utilização da aplicação protegida.

Os cadeados acompanham o software quando este é adquirido, devendo ser adicionados ao computador para que seja possível usar a aplicação neste equipamento. Esta é uma solução de proteção realmente segura, pois sem o cadeado torna-se praticamente impossível a utilização do software. Além disto, os cadeados não geram nenhuma forma de atividade de suporte para o administrador da rede/sistema, já que, uma vez instalados, estes dispositivos passam a atuar sem qualquer necessidade de configuração. Porém, os cadeados penalizam o nível de satisfação dos usuários ao ser criada uma redução na flexibilidade no uso do produto [Greguras 94]. Por exemplo, o parque computacional de uma determinada corporação pode ter modelos de PCs com algumas incompatibilidades de hardware que impossibilitem o compartilhamento do mesmo tipo de cadeado. Isto iria restringir a utilização da aplicação protegida por um determinado cadeado a certos

modelos de máquinas. Além disso, os cadeados teriam que se adaptar aos avanços tecnológicos dos componentes de hardware. Por exemplo, como implementar algo similar ao orifício na superfície magnética de um disquete em um CD? Devemos considerar também a possibilidade do surgimento de novos padrões de interfaces de hardware (seriais, paralelas, etc.).

Nada impede que a aplicação protegida pelo cadeado esteja instalada em diversas máquinas, porém apenas a quantidade delas correspondente à quantidade de cadeados disponíveis funcionará. Devido a este fato, torna-se comum a grande rotatividade dos cadeados entre os usuários dentro de uma corporação, causando, entre outras coisas, a danificação e até a perda do dispositivo. Este recurso nunca foi bem aceito por usuários finais, principalmente nos EUA, o que provocou uma descontinuidade por parte dos fabricantes de software na adoção desta alternativa como solução para a pirataria [Élan 95].

1.1.3 Proteção por Software

O controle da utilização de aplicações através de software é especificado e implementado pelo desenvolvedor, como parte integrante da aplicação, o qual iremos tratar aqui como **módulo de proteção**. Cada fabricante implementa sua própria solução. A política de proteção pode seguir diferentes abordagens, em função do ambiente operacional onde elas serão executadas. Seguindo este enfoque, faremos nesta seção considerações sobre estas soluções.

1.1.3.1 Proteção em Ambiente Centralizado

Ambiente mono-usuário - Para coibir o uso indevido de software nestes ambientes, o módulo de proteção permite a ativação do produto, ou partes dele, através de chaves de ativação.

Na proteção baseada em chaves de ativação de software, no ato da instalação do produto é solicitada uma seqüência especial de caracteres fornecida pelo desenvolvedor, sem a qual não será possível a conclusão da instalação. As chaves de ativação permitem também que uma cópia de demonstração enviada a um cliente possa se tornar cópia comercial, sem que, para isto, o cliente tenha que receber nova mídia contendo a versão definitiva.

O problema com as chaves de ativação é que estas, na maioria dos casos, são função apenas do número de série do produto. Desta forma, para burlar seu mecanismo de segurança, basta que uma nova instalação da aplicação seja feita em outra máquina, onde a mesma chave de ativação poderá ser fornecida para habilitar seu uso.

O esquema da chave de ativação torna-se mais seguro se esta for função de informações mais específicas da máquina do cliente. Por exemplo, ela pode ser função do numero de série do produto, da quantidade de memória disponível na

máquina, do número de série do disco rígido e da velocidade da CPU. Neste caso, obviamente, o cliente teria que entrar em contato com o desenvolvedor e fornecer estes dados para obter a chave. Além do mais, toda vez que um dos parâmetros mudasse, seria necessário obter uma nova chave. Há poucas empresas que utilizam este esquema. Um ponto importante é que este procedimento não se adequa bem a software comercializado em bancas de revistas, lojas e supermercados especializados em informática, também conhecidos como software de prateleira, justamente devido à necessidade de manutenção de um serviço de atendimento aos clientes para cálculo de chaves de ativação.

Ambiente multi-usuário - Além de proceder de maneira idêntica à situação dos ambientes mono-usuário, o módulo de proteção ainda pode permitir que um certo número de usuários possa executar simultaneamente uma aplicação. Aqui apresentaremos uma solução existente para módulos de segurança em ambiente Unix baseada em filas de mensagens, que são objetos existentes no núcleo do Unix.

A idéia é bastante simples e eficiente. O módulo de segurança irá controlar a quantidade de usuários executando a aplicação protegida baseado na quantidade de mensagens armazenadas na fila. O conteúdo das mensagens é o nome do terminal onde o usuário da aplicação se encontra. Portanto, se houver aplicações sendo usadas à partir dos terminais *ttya* e *ttyb* então haverá 2 mensagens enfileiradas: uma com o texto *ttya* e outra com o texto *ttyb*.

Quando uma aplicação entra em execução, ela descobre o nome de seu terminal e acessa a fila de mensagens para remover qualquer possível mensagem contendo o nome deste terminal. Isto é interessante para limpar qualquer mensagem deixada no passado por algum motivo (por exemplo, aplicação teve um fim anormal). A aplicação ativa uma *system call* para verificar o número de mensagens na fila, decidindo se vai permanecer executando ou não. Se sua execução for prosseguir, ela insere uma mensagem na fila com o texto apropriado. Como as operações de contagem de mensagens da fila e colocação de uma mensagem não são atômicas no tempo, pode haver uma situação de corrida que levará à ativação de mais cópias da aplicação do que o limite estabelecido. Isto é resolvido com o travamento da fila de mensagens no início da contagem, sendo esta liberada no final da transação.

Note que o objeto fila de mensagens foi escolhido por não ser um objeto persistente. Portanto, caso o sistema falhe, as filas de mensagens existentes simplesmente desaparecem, o que não aconteceria com arquivos, por exemplo [Sauvé 96].

Esta solução é bastante interessante, porém, aplicável apenas a alguns tipos de ambientes centralizados.

1.1.3.2 Proteção em Ambiente de Rede

Com a rápida proliferação das redes de computadores, os usuários estão rapidamente migrando suas aplicações *stand-alone* para ambientes distribuídos, o que requer novas políticas de controle de utilização de software, de forma a suprir as necessidades de clientes e produtores/vendedores.

Em um ambiente de rede, o esquema de proteção das aplicações tem que ser muito mais eficaz, pois a cada dia que passa, fica mais forte a idéia de que o uso ilegal de software possui "localidade de referência". Isto significa que a maior concentração de cópias piratas de programas está em torno das respectivas cópias legais destes produtos. Empresas que instalam cópias legais de software tendem a disseminar este produto de forma criminosa por toda a sua base computacional instalada. Uma vez que esta base computacional esteja interligada por uma rede local, deve ser possível a detecção destas cópias irregulares de software.

O deslocamento para a computação cliente/servidor é um dos maiores marcos da indústria de software nos dias atuais. A computação distribuída está criando um ambiente computacional global, onde novas abordagens de controle de utilização de software se fazem necessárias. É neste contexto que surge um novo paradigma para a repressão à pirataria: o gerenciamento de licenças.

Em linhas gerais, o serviço de gerenciamento de licenças consiste em um sistema distribuído que controla a utilização de produtos de software segundo os termos acordados entre seus fornecedores e compradores, termos ou regras estas conhecidas como licenças. Este sistema distribuído funciona basicamente da seguinte forma: para que uma aplicação possa executar, esta pede autorização a uma entidade conhecida como servidor de licenças. Este é responsável por liberar ou negar licenças de execução, utilizando como critério políticas bem definidas de como as aplicações devem ser utilizadas. No próximo capítulo abordaremos com mais detalhes esse mecanismo de coibição de uso ilegal de software.

1.2 Objetivo e Organização do Trabalho

Como visto na seção anterior, existem várias alternativas para a coibição do uso ilegal de software, sendo estas classificadas basicamente em proteção legal, proteção por hardware e proteção por software. A proteção legal prevê sanções jurídicas contra aqueles que violarem os termos de utilização de um produto de software. Já a proteção por hardware é baseada no acoplamento de dispositivos adicionais ao computador para que assim o software protegido possa ser utilizado. A alternativa de proteção por software é baseada em chaves de ativação, solicitadas no ato da instalação do produto. A maior parte destas alternativas, entretanto, não tem se mostrado realmente eficaz no combate à pirataria. A proteção legal necessita de elementos de fiscalização, que garantam sua aplicação. A proteção por hardware não foi bem aceita pelos usuários por limitar a flexibilidade

no uso do software protegido. Por sua vez, a proteção por software baseada em chaves de ativação é relativamente fácil de ser burlada.

Vimos ainda que, embora seja ineficaz a aplicação da proteção legal de forma isolada, esta pode ser utilizada em conjunto com outras modalidades de proteção. De maneira geral, a atribuição de fiscal para a proteção legal é dada aos administradores de sistema/redes dos ambientes computacionais localizados nos compradores de software. Porém, o crescimento das corporações tem dificultado a ação eficiente dos administradores, que passaram a necessitar de ferramentas de auxílio ao policiamento de seus usuários.

Em conjunto com a proteção legal, um paradigma de proteção de software que vem ganhando bastante força e popularidade entre desenvolvedores e administradores é o gerenciamento de licenças. Esta solução vem ganhando importância no mercado de software por ser a solução técnica para a proteção da propriedade intelectual que apresenta maior flexibilidade, atuando, inclusive, como uma importante ferramenta nas estratégias de *marketing* e distribuição dos produtos [Élan 95].

Existem várias formas e critérios para limitar o acesso de um usuário a uma aplicação. Através do gerenciamento de licenças, é bastante simples configurar políticas de licenciamento, bem como reconfigurá-las, quando necessário. Uma vez que a utilização de uma determinada aplicação está sendo controlada por um serviço de gerenciamento de licenças, todas as ativações desta aplicação devem necessariamente passar pela aprovação do gerenciador de licenças. Desta forma, importantes estatísticas acerca da utilização de um produto de software (quantidade de ativações em um determinado intervalo de tempo, horários de pico, demanda de utilização) podem ser produzidas, possibilitando um *feedback* importantíssimo tanto para administradores quanto para os próprios desenvolvedores, no que diz respeito à correta atribuição de preços aos seus produtos de software (valor percebido).

Esta flexibilidade apresentada pelo gerenciamento de licenças, porém, tem o seu preço. Além da necessidade da configuração de políticas de licenciamento para as aplicações protegidas, o gerenciamento de licenças pode apresentar problemas de suporte que devem ser resolvidos pelo administrador do sistema/rede, e que podem aumentar sobremaneira a carga de trabalho deste já normalmente sobrecarregado profissional. Sob o ponto de vista do desenvolvedor que adota este tipo de proteção para suas aplicações, há a necessidade de se manter uma equipe de antendimento aos clientes, que dê suporte à configuração de políticas de licenciamento (geração de chaves de ativação baseadas em informações do ambiente computacional do cliente, campos de *checksum* para linhas de arquivos de licenças, etc.). Isto não é adequado para pequenos produtores de software, que necessitam ter baixos custos operacionais, para que seus produtos tenham preços competitivos e acessíveis.

Como forma de contornar esta limitação, estamos propondo em nosso trabalho um serviço de gerenciamento de licenças cuja meta é, além de proteger as aplicações contra uso ilegal, eliminar os problemas de suporte administrativo e técnico, comuns em ferramentas deste tipo, ou seja, ser tolerante a faltas[1], auto-configurável e livre de suporte. Uma vez instalado, nosso gerenciador de licenças deverá dispensar qualquer atenção do administrador para o seu bom funcionamento.

O restante deste trabalho está organizado da seguinte forma: no capítulo 2 apresentaremos de maneira detalhada a solução de proteção baseada no serviço de licenciamento, abordando os principais produtos existentes no mercado e os problemas de suporte que estes podem apresentar.

Nossa solução de licenciamento, denominada **Bouncer**, será apresentada no capítulo 3, onde serão discutidos seus serviços, sua arquitetura e requisitos necessários para sua implementação.

O capítulo 4 trará uma análise dos paradigmas de comunicação usados pelos processos que compõem o Bouncer.

Finalmente, as conclusões do trabalho e sugestões para trabalhos futuros serão apresentadas no capítulo 5.

---

[1] Neste trabalho estamos usando a terminologia proposta em [Lemos-Veríss 91], que por sua vez se baseia na que é proposta em [Lamprie 89], traduzindo os termos *fault*, *error* e *failure* como falta, erro e falha, respectivamente.

2

2 Serviços de Licenciamento

O termo licença de software representa o direito e as regras de utilização de um determinado software disponível em um computador. Um **serviço de licenciamento** ou **gerenciador de licenças** é um sistema que garante o cumprimento da licença de software em ambiente multi-usuário ou de redes de computadores. Tal serviço vem ganhando cada vez mais força junto aos desenvolvedores, principalmente devido à sua flexibilidade. Além de garantir proteção à propriedade intelectual dos produtores de software, os gerenciadores de licença estão sendo uma importante ferramenta nas estratégias de mercado traçadas pelas empresas do setor [Élan 95], como veremos mais adiante.

Na maioria dos casos, o serviço de licenciamento é baseado no modelo cliente-servidor e oferece uma interface de programação que permite ao desenvolvedor de software integrá-lo aos seus produtos. Com isto, a política de licenciamento do produto irá ficar isolada de sua funcionalidade, em uma camada de software específica e bem definida.

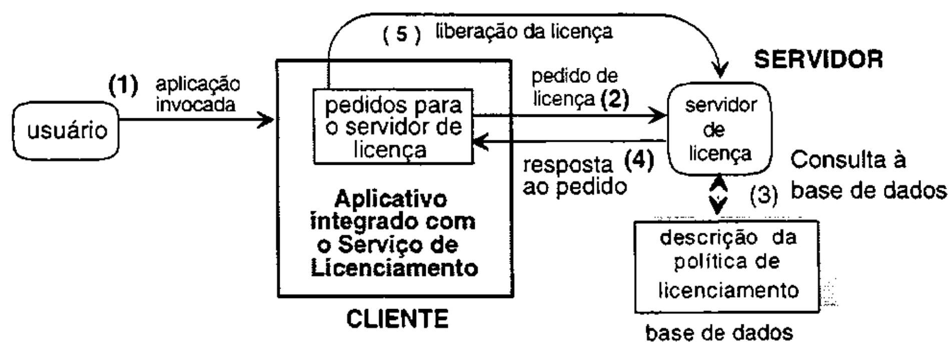


Figura 2.1 - Uso de aplicativos integrados com um serviço de licenciamento

A figura 2.1 mostra, de forma simplificada, as operações que ocorrem quando um usuário executa aplicativos administrados por um serviço de licenciamento. Os passos realizados são:

1. O usuário executa a aplicação.
2. A aplicação solicita ao servidor de licenças uma licença para seu uso.
3. O servidor verifica se é possível atender ao pedido de licença, acessando uma base de dados que descreve a política de licenciamento e contém as informações relevantes sobre as licenças em uso.
4. Se é possível atender ao pedido de licença, o servidor responde positivamente ao cliente, que avisa à aplicação para prosseguir sua execução. Caso não seja possível, a atitude a ser tomada dependerá do desenvolvedor (finalizar a aplicação, alertar o usuário através de uma mensagem e prosseguir a execução, etc.).
5. Quando a aplicação finaliza, esta informa ao servidor a liberação da licença.

Existem diversas aplicações que utilizam serviço de licenciamento. Como exemplos, podemos citar desde ferramentas de desenvolvimento para ambiente Unix (ex.: Compiladores C/C++ [SunOS 90]), ferramentas de gerência de rede (OpenView [OpenView 96]) até ferramentas de produtividade pessoal (ex.: Lotus 123 para Unix [Lotus 90]).

Podemos apontar vários ganhos, tanto para usuários quanto para produtores/vendedores de produtos de software, com o uso de serviço de licenciamento:

- Redução significativa do uso de cópias não autorizadas de software.
- Extrema facilidade para o cumprimento dos termos de utilização do produto.
- Uma vez que o uso das aplicações apenas será permitido segundo seus respectivos contratos de licenciamento, o administrador do sistema não irá precisar dedicar parte do seu tempo para monitorar a utilização de programas, verificando se os mesmos estão sendo legalmente utilizados.

Além disso, a utilização do gerenciamento de licenças também pode ser visto como uma estratégia de mercado, não sendo usado apenas com propósitos de proteção da propriedade intelectual. Isto se dá da seguinte forma:

- Produtores/vendedores de software podem desenvolver novos modelos de atribuição de preço aos seus produtos, devido à flexibilidade em implementar políticas de utilização de software que reflitam as reais necessidades dos usuários. Através do gerenciamento flexível de licenças, é possível mensurar a real demanda de utilização para produtos de software, inclusive de características e módulos específicos destes. Isto irá fornecer

subsídios aos produtores para o estabelecimento de preços (valor subjetivo do produto - importância do produto para o cliente).

- O usuário paga apenas pelas suas necessidades de uso.
- A distribuição eletrônica de software (através da Internet, por exemplo), irá ser cada vez mais popularizada graças às políticas de licenciamento utilizadas em cópias para demonstração (número de acessos, data de expiração, tempo de uso, etc.). Note que o gerenciamento de licenças pode permitir a conversão de uma cópia de demonstração para uma cópia comercial, tornando este procedimento bastante simples.
- Com a coexistência de várias plataformas distintas de *hardware* em redes heterogêneas, os clientes, ao comprarem um determinado software, não querem utilizar políticas de licenciamento para máquinas específicas. Em vez disso, eles desejam que os fabricantes forneçam uma solução de controle global do uso da aplicação, independente da plataforma onde esta seja executada. Esta flexibilidade irá garantir uma utilização mais efetiva da aplicação, uma vez que é possível que licenças existentes para uma determinada plataforma não estejam no momento sendo utilizadas e usuários desta mesma aplicação, em uma versão específica para outra máquina, não estejam encontrando licenças suficientes para trabalhar. Com o gerenciamento global das licenças, torna-se mais fácil para o administrador do sistema dimensionar o número de licenças adequado para atender aos seus usuários, uma vez que este não precisa se preocupar em distribuir as licenças por plataforma disponível. A compra de novas plataformas de *hardware* pelos clientes não necessariamente deverá implicar em ter que modificar os termos de licenciamento junto ao desenvolvedor (a não ser que se queira aumentar o número de licenças para a aplicação), bastando solicitar deste a respectiva versão da aplicação e instalá-la na nova máquina.

Veja um quadro resumo com as vantagens oferecidas pelo uso dos gerenciadores de licenças.

Vantagens Proporcionadas pelos Gerenciadores de Licenças	Beneficiário
Facilidade no cumprimento dos termos de licenciamento de software.	administrador/ usuário
Ferramenta de auxílio na fiscalização do uso legal de software.	administrador
Registro da demanda de utilização de aplicações.	administrador/ usuário
Conversão de cópias de demonstração em cópias definitivas.	administrador/ desenvolvedor
Auxílio no estabelecimento de preços dos produtos.	desenvolvedor/ usuário
Controle de utilização de software multi-plataforma.	desenvolvedor/ usuário

Tabela 2.1 - Vantagens do uso de gerenciadores de licença.

2.1 Definindo Políticas de Licenciamento

Existem vários atributos que podem ser usados na definição da política de licenciamento. Vejamos a descrição de alguns dos mais importantes deles:

- **Quantidade de ativações concorrentes:** Indica o número de usuários que podem usar uma mesma aplicação simultaneamente. Potencialmente, qualquer usuário, de qualquer máquina da rede, pode obter licença para uso de uma aplicação, contanto que o número máximo de licenças não seja ultrapassado. Normalmente, quando isso ocorre, o usuário recebe um aviso de que não existe licença disponível e, em seguida, é instruído a tentar, posteriormente, uma nova execução da aplicação. Uma outra possibilidade é colocar os pedidos dos usuários que não obtiveram licença em uma fila de espera, liberando-os oportunamente, quando houver disponibilidade para a execução da aplicação. Esta é uma das principais políticas de licenciamento utilizadas.
- **Conjunto de compartilhamento:** Permitem o compartilhamento de uma licença entre várias aplicações ou um certo número de ativações de uma aplicação com a contabilização de apenas uma licença. Este atributo pode ser usado, por exemplo, para aplicações usadas em ambiente gráfico baseado em janelas. Desta forma, o usuário poderá ter várias cópias de uma aplicação em um mesmo terminal gráfico, consumindo uma única licença.
- **Quantidade de ativações acumuladas:** Indica o número pré-determinado de vezes que o usuário pode ativar a aplicação. Toda vez que a aplicação é executada, o servidor de licenças decrementa o número máximo de ativações permitidas. Isto é feito até que este número máximo de ativações permitidas chegue a zero. A partir desse ponto, o usuário não consegue mais obter licença para executar a aplicação. Este atributo é adequado para cópias de demonstração.

- **Data de expiração:** Estabelece uma data de validade para a utilização da aplicação. Uma vez expirada esta data, não mais será permitida a ativação da aplicação. Este atributo também é utilizado para cópias de demonstração.
- **Reserva de execução:** Especifica os nós da rede, usuários ou grupos de usuários habilitados a ativar a aplicação. O próprio servidor de licenças normalmente utiliza este atributo. Em serviços de licenciamento baseados no modelo cliente-servidor, o servidor de licenças é instalado em nós previamente especificados.
- **Anotações de Licença:** Permitem definir licenças para diferentes módulos das aplicações. Por exemplo, um editor de texto que oferece um corretor ortográfico como módulo opcional independente. É possível definir uma anotação de licença para que o serviço de licenciamento trate o editor de texto e o corretor ortográfico como uma mesma aplicação. Anotações de licença são definidas pelo vendedor da aplicação e incluídas como parte das licenças, quando da instalação dos produtos.

É importante observar que todos os atributos são ortogonais, podendo ser combinados de forma a estabelecer políticas de licenciamento mais sofisticadas. Por exemplo, uma aplicação pode possuir licença de ativação para 10 usuários, simultaneamente: 2 para nós específicos (uma das quais expira em 15 de julho) e 8 para qualquer nó da rede (3 das quais com direito a apenas 10 ativações e 2 para uso exclusivo do diretor da empresa e do administrador da rede). Note que este exemplo utiliza de maneira exagerada a característica de ortogonalidade dos atributos com fins ilustrativos, não sendo encontrado na vida real. Um exemplo de política híbrida mais próxima da realidade seria o licenciamento de um módulo editor HTML que funciona acoplado a um *browser* WWW. Uma possibilidade é definir uma licença que indique que este *plug-in* só pode ser utilizado simultaneamente por 3 dos usuários do *browser* que pertencerem à equipe de desenvolvimento *web*; cada um destes usuários, porém, pode abrir em sua interface gráfica quantos editores necessite, sem que seja contabilizada mais que uma licença por usuário.

2.2 Principais Licenciadores Existentes no Mercado

Nesta seção serão apresentados os principais serviços de licenciamento existentes no mercado.

2.2.1 FlexLM

Este é um produto distribuído pela empresa *GLOBEtrotter*. O FlexLM (*Flexible Licence Manager*) é um sistema distribuído para o gerenciamento de licenças em

redes heterogêneas. O FlexLM adota o modelo cliente-servidor e é composto pelos seguintes elementos [FlexLM 96] [FlexTO 96]:

- lmgrd (*daemon*[1] de licenças);
- Vendor *daemon* (*daemon* do fabricante);
- Arquivo de licenças;
- API[2] cliente (FLEXlm client library).

lmgrd - Este é um módulo executável que concentra funcionalidades de servidor de licenças. Uma vez executado, o lmgrd se conecta a uma porta TCP/IP bem conhecida e desempenha basicamente três funções principais: a de mapear portas TCP/IP para os *daemons* dos fabricantes (detalhado no próximo tópico), inicializá-los e, caso venham a falhar, reinicializá-los. Este servidor pode estar replicado em até três máquinas, pré-definidas em um arquivo de licenças (veja tópico adiante).

Vendor daemon - Existe um processo específico para cada fabricante, responsável pelo controle das licenças dos seus respectivos produtos. O daemon do fabricante fornece informações sobre quantas licenças estão sendo consumidas e quem as está utilizando.

Arquivo de licenças - Este é um arquivo do tipo texto, contendo informações de controle para a aplicação cliente (o *host* e a porta TCP/IP onde se localiza o lmgrd), para o lmgrd (informações sobre *daemons* dos fabricantes) e para os daemons de fabricantes (descrição das licenças dos produtos). Cada linha de dados referente aos produtos licenciados, chamada FEATURE, contém uma chave construída com base nos dados nela contidos, como forma de autenticar estes dados, de modo que qualquer alteração no conteúdo da linha irá causar erro na reconstrução da chave de segurança. O arquivo de licenças possui as seguintes palavras-chave:

SERVER - Esta linha contém o nome do servidor (*hostname*), onde os clientes irão procurá-lo, o *hostid* (número que identifica de maneira única uma CPU) e uma porta do sistema operacional onde o processo servidor irá esperar ser contactado por aplicações clientes. As informações *hostname* e porta podem ser alteradas pelo administrador do sistema sem nenhum problema, o que não acontece com o *hostid*. Qualquer alteração neste campo irá invalidar todas as licenças contidas no arquivo, uma vez que as chaves de segurança de cada linha são construídas baseadas também no *hostid*. Um arquivo de licenças pode conter até três linhas SERVER.

DAEMON - Nesta linha é especificado o nome do *vendor daemon*, bem como a localização de seu código executável.

---

[1] Um *daemon* é um processo que executa desconectado de terminais (em *background*).

[2] API (*Application Program Interface*) - Interface de programação.

FEATURE - Linha onde são descritas as licenças. As informações nela contidas não podem ser modificadas, pois são geradas pelo desenvolvedor. São elas:

- nome - o nome da aplicação.
- *daemon* - o nome do *vendor daemon* que irá servir esta licença.
- versão - a versão da aplicação cuja licença está sendo descrita.
- data de expiração - a data limite de validade da licença, após a qual a respectiva aplicação não poderá mais ser executada; caso o ano seja 0, a licença nunca irá expirar.
- número de licenças - quantidade de licenças concorrentes disponíveis para a aplicação; caso este campo tenha valor 0, as licenças serão tidas como ilimitadas, não necessitando de servidor para controlar esta licença; isto é usado em licenças do tipo data de expiração ou reservadas por máquina.
- chave - chave de segurança gerada a partir dos dados contidos na linha FEATURE e no campo *hostid*, como comentado anteriormente.
- *string* do fabricante - seqüência de até 64 caracteres entre aspas duplas ("") que pode ser utilizada como parâmetros de entrada para o *daemon* do fabricante.
- *hostid* - usado opcionalmente, para indicar que a licença está restrita a uma máquina em particular.

INCREMENT - Esta é uma maneira de acrescentar mais licenças a uma determinada aplicação. Cada linha deste tipo deverá estar relacionada a uma FEATURE, através dos campos nome, versão e *hostid* (se houver).

UPGRADE - Possui os mesmos dados que as linhas FEATURE ou INCREMENT, com o acréscimo do campo versão_origem, que deverá vir antes do campo versão. Esta linha redefine um certo número de licenças de uma versão antiga de uma aplicação, convertendo tais licenças para uma versão mais atual do software.

PACKAGE - Define grupos de linhas FEATURE ou INCREMENT que correspondem a pacotes de software com vários módulos independentes.

Exemplo de um arquivo de licenças

```
SERVER excellent_server          17007ea8       1700
DAEMON acmed /etc/acmed
FEATURE acme_appl acmed 1.000 01-jun-1997 10 1EF890030EABF324 ""
FEATURE acme_app2 acmed 1.000 01-jun-1997 10 0784561FE98BA073 ""
```

O arquivo de licenças acima especifica que o servidor de licenças de nome *excellent_server* está executando em uma máquina cujo *hostid* é 17007ea8, esperando conexões de aplicações clientes na porta do sistema operacional de número 1700. O *daemon* acmed, localizado no diretório /etc, é o responsável por servir licenças para as aplicações acme_appl1, versão 1.000 e acme_appl2, versão

1.000. As linhas FEATURE descrevem as políticas de licenciamento para ambas as aplicações, que possuem 10 licenças de uso concorrente, as quais expiram em 01 de junho de 1997. O arquivo de licenças deve estar disponível tanto nas máquinas onde executam os clientes quanto o(s) servidor(es), seja localmente, seja via um sistema de arquivos de rede (*Network File System* - NFS).

API cliente - Esta é a componente cliente do FlexLM, que é ligada à aplicação, responsável pela comunicação com o *daemon* lmgrd para solicitar a licença de execução. A tabela abaixo traz um resumo das funções disponíveis na interface de programação do FlexLM

Função	Descrição
lp_checkin()	Solicita uma licença de execução.
lp_heartbeat()	Testa constantemente a existência do servidor de licenças (*vendor daemon*). Em caso de queda do servidor e posterior re-inicialização, esta função é responsável por re-conectar a aplicação ao *daemon.*
lp_errstring()	Obtém a descrição do último erro ocorrido.
lp_checkout()	Libera uma licença de execução.

Tabela 2.2 - Funções disponíveis na API do FlexLM.

O processo de solicitação de licenças

1. A aplicação cliente procura pelo arquivo de licenças e dele obtém o host e a porta do servidor de licenças (lmgrd).
2. O cliente estabelece uma conexão com o lmgrd e informa a este com que *daemon* de fabricante este deseja conversar.
3. O lmgrd determina qual a máquina e porta onde o *daemon* do fabricante se encontra e envia esta informação de volta ao cliente.
4. O cliente estabelece uma conexão com o *daemon* específico do fabricante, solicitando uma licença.
5. O *daemon* do fabricante verifica se ainda há licenças disponíveis para a aplicação, concedendo ou negando o pedido do cliente.
6. O módulo cliente da aplicação entrega o resultado do pedido de licença à aplicação.

Parâmetros configuráveis do FlexLM

- a localização do arquivo de licenças;
- a localização de todos os seus módulos executáveis;
- a localização dos arquivos de log;
- a porta TCP/IP usado pelo processo lmgrd.
- o nó (ou nós, em caso de replicação do serviço) em que o lmgrd irá executar.

Os tipos de licença suportados pelo FlexLM são: quantidade de ativações concorrentes, reserva de execução (*nodelocked*), data de expiração e anotação de licenças.

O preço do FlexLM para a primeira plataforma, com direito a produzir um número ilimitado de licenças para seus compradores, é de US$ 40,000, sendo reduzido a cada plataforma adicional. A *GLOBEtrotter* possui uma modalidade especial de atribuição de preços direcionada para desenvolvedores que ela classifica como "*emerging companies*" (pequenas empresas em ascensão), que consiste em cotar o FlexLM de acordo com as vendas anuais do desenvolvedor. A faixa de preços, neste caso, varia a partir de US$ 1,000 [Miller 96].

2.2.2 ÉlanLM

Desenvolvido pela *Élan Computer*, é um pacote de software que controla a utilização de aplicativos em ambientes de rede ou multi-usuário. É baseado no modelo cliente-servidor, sendo o gerente de licenças o servidor e as aplicações os clientes. O ÉlanLM é composto por [ÉlanLM 95]:

- elmd (*daemon* servidor de licenças);
- arquivo de licenças;
- API cliente;

elmd - Módulo executável que concentra toda a funcionalidade de servidor. O elmd possui um esquema de chave, conhecida como *host-lock*, que só lhe permite executar na máquina para qual a chave foi gerada. Ou seja, o próprio módulo servidor de licenças é licenciado através de uma chave de ativação, específica para a máquina onde irá executar. Esta chave baseia-se em informações como endereço IP, número da CPU (em PROM), *hostname*, endereço Ethernet, endereço IPX ou número serial NetWare.

O servidor é único por fabricante e, para oferecer tolerância a faltas, pode ser replicado em vários *hosts* da rede, não havendo um limite máximo para o possível número de réplicas. Quando é usada a replicação do servidor de licenças, na falha do *daemon* principal ou de seu *host*, um outro servidor será automaticamente inicializado e as licenças ativas para ele transferidas. O elmd está apto a monitorar aplicações protegidas que tiveram um fim anormal, liberando automaticamente suas licenças.

Arquivo de licenças - Onde são armazenados o código do *host* e as chaves fornecidas ao elmd. As licenças para as aplicações também são codificadas em forma de chaves. Nelas estão contidas todas as informações sobre a política de licenciamento de uma determinada aplicação. Estas chaves são geradas pelo desenvolvedor através de uma ferramenta de administração (Elmkey), utilizando os atributos de licenciamento (nome da aplicação, número de licenças, data de expiração) e o código do *host* (comentado adiante) onde o servidor de licenças irá

executar. Esta última informação é fornecida pelo cliente. O arquivo de licenças é criptografado.

API cliente - Funcionalidade cliente do ÉlanLM, acoplada ao código do desenvolvedor. A tabela a seguir apresenta um resumo das funções disponíveis na interface de programação do ÉlanLM.

Função	Descrição
elm_init()	Função chamada no início da aplicação, responsável por estabelecer conexão com o servidor de licenças.
elm_heartbeat()	Instala um mecanismo (heartbeat) que notifica o servidor de licenças caso a aplicação falhe.
elm_getlicense()	Solicita uma licença de execução.
elm_bye()	Desconecta a aplicação do servidor, liberando a licença por ela consumida.

Tabela 2.3 - Funções disponíveis na API do ÉlanLM.

Criação de licenças

Para que uma aplicação esteja habilitada a executar, segundo a política de licenciamento para ela adotada, esta inicialmente deverá ser instalada. Após a instalação, devem ser seguidos os seguintes passos:

1. geração do código do servidor - O usuário final deve utilizar uma ferramenta de administração (Elmadmin) que, baseado em informações sobre o host, gera um código que identifica de maneira única aquela CPU. Este código deve ser informado ao fabricante (via e-mail, fone ou fax), para que este gere a chave de licenças. Após a geração do código do *host*, Elmadmin irá aguardar o fornecimento da chave de licenças do produto.

2. geração da chave de licenças - O fabricante gera a chave de licenças através da ferramenta Elmkey, entrando com as informações sobre a política de licenciamento e o código do *host*, como explicado acima. A chave gerada é enviada ao comprador.

3. instalação da chave de licenças - O comprador, finalmente entra com a chave de licenças via Elmadmin, que já está aguardando esta informação, como mencionado no passo 1. A partir disto, a aplicação licenciada está habilitada a ser executada.

Todos os tipos básicos de licença são suportados. O preço do ÉlanLM para a primeira plataforma fica em torno de $7,500 [Lovett 96], sendo reduzido para as plataformas adicionais.

2.2.3 iFOR/LS

Esta é a solução de licenciamento da Gradient Technologies, baseada no modelo cliente-servidor e no *Network Computing System* (NCS) [IBM 96]. O

paradigma de comunicação entre processos clientes e o servidor de licenças adotado é o de chamada a procedimentos remotos (RPC) [Birrell 84]. O iFOR/LS suporta licenças reservadas (*nodelocked*) e licenças concorrentes. Seus componentes são [Gradient 95]:

- netlsd *(daemon* servidor de licenças).
- serviço de nomes.
- arquivo de licenças.
- ferramentas de administração (iFOR/LS Administrator's Runtime Kit).
- API cliente.

netlsd - Módulo executável que concentra as funções de servidor de licenças. Este *daemon* pode ser instalado em um ou mais nós da rede. Para instalar o netlsd em uma máquina, é preciso que nela esteja previamente instalado o iFOR/LS Administrator's Runtime Kit.

O netlsd não é necessário para o modelo de licenças com reserva de execução por nó da rede (*nodelocked*), apenas para o modelo de licenciamento quantidade de ativações concorrentes.

Serviço de Nomes - Com o objetivo de localizar o serviço de licenciamento na rede, o iFOR/LS se utiliza da infra-estrutura de *location broker* do NCS, composta por dois *daemons*: **glbd** (*global location broker daemon*), responsável pela manutenção de uma base de dados contendo todos os serviços oferecidos na rede (este daemon pode estar em um ou mais nós da rede) e o **llbd** (*local location broker daemon*), presente em todos os nós da rede que provêm algum serviço, inclusive naqueles onde o glbd executa. Seu objetivo é gerenciar a comunicação entre o glbd e os servidores. Para o iFOR/LS, portanto, deve haver disponível a seguinte infra-estrutura do serviço de nomes:

- um ou mais nós da rede rodando o glbd;
- para cada nó onde o glbd esteja em execução, o llbd também deverá estar ativo;
- para cada nó onde o netlsd esteja em execução, o llbd também deverá estar ativo;

O NCS, juntamente com o netlsd são inicializados em tempo de *boot*, através de um script de inicialização, procedimento, aliás, comum a todos os produtos discutidos neste capítulo.

Arquivo de licenças - Arquivo utilizado para controlar as licenças do tipo reservadas. Para o uso deste tipo de licenciamento, onde a execução da aplicação fica restrita a uma CPU específica, não é necessária a presença do *daemon* servidor. Basta que seja instalada a aplicação cliente na respectiva máquina e criado, pelo super-usuário (root), um arquivo do tipo texto (/usr/lib/netls/conf/nodelock). Este arquivo é composto por vários blocos com o seguinte formato:

```
# Netscape 3.0 Gold expires 12/25/98
4ca0f7ea1000.0d.00.02.1a.9a.00.00.00 7dp63x23jqevxenx7ccr4ejmwn "" "2.0"
```

A primeira linha de cada bloco é composta por um caracter de comentário (#), o nome do produto licenciado e sua data de expiração. A linha seguinte traz o identificador do fabricante, a *password* da licença, um campo opcional para anotações de licenças (entre "") e, finalmente, a versão do produto.

A *password* da licença é gerada pelo fabricante da aplicação, baseada em um identificador da máquina habilitada a usar a licença. O administrador do sistema do usuário final deverá obter o identificador do host do servidor de licenças, utilizando para isto uma ferramenta de administração, comentada a seguir. O identificador obtido deverá ser informado ao fabricante, que irá gerar as respectivas *passwords* de licença e enviá-las ao cliente. Na password já está embutido o número de licenças concorrentes disponíveis para a aplicação.

Ferramentas de administração - São aplicativos que possibilitam modificar informações sobre produtos e seus fornecedores na base de dados do servidor de licenças. As principais ferramentas são:

ls_targetid - É utilizada para determinar o identificador do *host* onde está instalado o servidor de licenças.

ls_admin - Ferramenta utilizada para instalar as licenças na base de dados do servidor.

API cliente - Funcionalidade cliente do iFOR/LS acoplada às aplicações durante o processo de compilação. As principais funções oferecidas ao desenvolvedor pela API do iFOR/LS estão descritas de forma sucinta na tabela 2.4.

Função	Descrição
netls_add_nodelocked	Adiciona uma nova *password* de licença ao arquivo de licenças reservadas da máquina onde esta função for ativada.
netls_request_license	Solicita uma licença de execução.
netls_request_and_wait	Solicita uma licença de execução. Caso não haja licença disponível, esta função irá enfileirar este pedido em cada um dos servidores disponíveis na rede.
netls_release_license	Libera uma licença de acesso concorrente.
netls_get_all_product_info	Retorna informações sobre licenças concorrentes de todos os produtos dos fabricantes especificados nos seus parâmetros.
netls_init	Localiza servidores de licenças para os programas de um determinado fabricante.

Tabela 2.4 - Funções disponíveis na API do iFOR/LS.

Instalando Licenças Concorrentes

Para cada nó rodando o netlsd, as respectivas licenças deverão ser instaladas. Como dito acima, isto é feito através da ferramenta de administração

ls_admin, que solicita os identificadores do fabricante e da aplicação protegida, além da password das licenças. Esta instalação deve ser feita pelo super-usuário. A partir deste momento, as aplicações estão habilitadas a executar concorrentemente.

Uma licença é associada a um servidor específico durante seu processo de instalação. Caso este servidor esteja fora do ar, nenhuma licença a ele associada poderá ser utilizada. Caso o netlsd seja instalado em mais de um nó da rede, as licenças disponíveis estarão divididas entre estes (cada servidor responde por um subconjunto das licenças).

O preço, por kit de desenvolvimento, do iFOR/LS é de US$ 8,000. O desenvolvedor deverá pagar ainda uma anuidade de US$ 7,500 pelos direitos de criação de licenças, o que dá ao desenvolvedor o direito de gerar um número ilimitado de licenças para seus compradores [Leveille 96].

2.3 Problemas do Serviço de Licenciamento

Apesar de toda a flexibilidade e eficiência oferecidas no combate ao uso ilegal de produtos de software, características estas bastante enfatizadas ao longo deste capítulo, os gerenciadores de licenças podem vir a apresentar alguns problemas de ordem operacional. Estes problemas geram a necessidade de suporte administrativo e técnico, geralmente prestado pelo administrador do sistema/rede. Passaremos agora a levantar os principais problemas de suporte.

Apesar dos servidores de licença controlarem as licenças automaticamente, estes necessitam da intervenção do administrador do sistema para atividades como configuração da política de licenciamento, criação e manutenção dos bancos de licença contendo dados sobre os produtos instalados, localização e inicialização dos *daemons* servidores, etc.

Além disso, é importante notar que uma característica essencial para qualquer serviço de gerenciamento de licenças é robustez, ou seja, um nível de confiabilidade que garanta uma alta disponibilidade do serviço. A busca por implementar esta característica, entretanto, acaba por gerar a maior parte das atividades de suporte para o administrador do sistema/rede. Todas as ferramentas de gerenciamento de licenças acima apresentadas procuram oferecer alta disponibilidade, geralmente com a replicação dos servidores de licenças em vários nós da rede. Vários problemas surgem também em virtude disto.

No FlexLM, por exemplo, o arquivo de licenças deve ser criado pelo desenvolvedor de aplicações e instalado pelo administrador do sistema do cliente. Este também é responsável por realizar manutenções no arquivo de licenças sempre que se faça necessária qualquer atualização na política de licenciamento. Quando um comprador de uma aplicação resolve ampliar o número de licenças

disponíveis em seu ambiente de rede ou adiar a data de expiração de uma cópia de avaliação de determinado produto, estas novas características terão que ser informadas ao gerenciador de licenças mediante a edição do arquivo de licenças. O mesmo deverá ser feito caso uma máquina que normalmente executa um servidor de licenças tenha que ser desativada para manutenção e este servidor instalado em outra máquina. É importante que fique bem entendido que os dados para estas atualizações são fornecidos pelo fabricante, uma vez que as chaves de cada linha deverão necessariamente ser re-geradas.

No caso do iFOR/LS, os problemas podem surgir na localização e inicialização dos *daemons* servidores. O iFOR/LS depende do serviço de nomes fornecido pelo *Network Computing Service* (glbd e llbd), que deve estar ativo para que as aplicações clientes possam localizar o servidor de licenças. Esta dependência pode gerar mais pontos de suporte para o administrador, que terá que monitorar o funcionamento deste serviço auxiliar. Em caso de falha dos *daemons* glbd, llbd e netlsd, o administrador deverá re-inicializá-los. Isto também ocorre com o FlexLM e ÉlanLM. Em casos de falha no serviço de licenciamento devido à queda dos *daemons* lmgrd e elmd, respectivamente, o administrador é o responsável tanto pela detecção da falha quanto por colocá-los em execução manualmente, muito embora réplicas destes servidores possam assumir temporariamente seus lugares (no caso de replicação do serviço de licenciamento).

No FlexLM, caso o servidor de licenças sofra alguma pane, as licenças em uso são automaticamente canceladas. Além disso, enquanto o servidor está desativado, não é possível obter licenças para os aplicativos que venham a querer executar.

Normalmente as aplicações são implementadas de tal forma que procedam a verificação periódica do estado do servidor. Caso este tenha falhado, a aplicação irá continuar sua execução, verificando periodicamente se o *daemon* já foi re-inicializado. Isto pode ser feito indefinidamente ou um certo número de vezes, quando a aplicação simplesmente irá encerrar sua execução, caso o serviço permaneça inativo. Como dito anteriormente, o daemon não é re-inicializado automaticamente.

Os serviços FlexLM e ÉlanLM mantêm servidores ou informações replicadas em outros nós da rede a fim de obter maior disponibilidade dos seus serviços. Mas, para isto, ou é imposto um número limitado de réplicas ou estas são específicas por host (um servidor não funciona em uma máquina diferente daquela para ele designada).

No FlexLM são permitidos apenas 3 servidores replicados. Uma vez inicializados, ao longo do tempo pode haver a falha de algum destes processos. Porém, pelo menos dois deles deverão estar em funcionamento para que haja liberação de licenças. Isto é chamado *quorum*. Quando a opção por replicação de servidores é feita, o *quorum* é uma maneira do FlexLM garantir que, no caso de falha do *daemon* servidor, haja pelo menos outro *daemon* apto a assumir seu lugar.

No ÉlanLM, há a necessidade de se vincular um processo servidor a uma máquina específica, através de uma chave de ativação gerada pelo fabricante (*node-lock*). Isto dificulta o processo de replicação dos *daemons*, uma vez que deve haver uma chave de ativação para cada réplica. Caso uma destas máquinas servidoras tenha que ser desativada provisoriamente para manutenção, por exemplo, é necessário que uma nova chave de ativação seja gerada pelo fabricante da aplicação protegida para que um *daemon* servidor de licenças possa ser migrado para outra máquina.

No iFOR/LS, quando mais de um servidor de licenças é instalado, o total de licenças disponíveis é rateado entre estes. Portanto, havendo falha em algum servidor, as licenças por ele controladas não estarão mais disponíveis até que este seja re-inicializado.

Caso o *host* onde uma aplicação protegida pelo FlexLM esteja executando falhar, nenhuma licença em execução naquele *host* será automaticamente liberada. Há a necessidade de se usar uma ferramenta de administração, chamada **lmremove**, para tal fim, o que também se constitui em uma atividade de suporte a ser executada pelo administrador.

Para cada nó da rede, deve existir no máximo um lmgrd em execução, como também um *daemon* de cada fabricante. Quando um lmgrd é cancelado, ele deve antes finalizar os respectivos *vendor daemons* a ele subordinados. Quando o lmgrd é cancelado de uma forma inesperada (ex., kill -9 no Unix), este não finaliza os *vendor-daemons*. Ao ser re-inicializado, o lmgrd re-inicializará todos os *daemons* de fabricante, podendo ocorrer múltiplos *daemons* do mesmo fabricante executando simultaneamente.

Em todos os gerenciadores apresentados, existe a necessidade dos desenvolvedores manterem pessoal de suporte para interagir com os compradores de seus produtos, exercendo atividades como geração de *passwords* de licenças e chaves de ativação de módulos servidores.

A seguir, mostramos um quadro resumo, confrontando os três gerenciadores de licença apresentados neste capítulo com os principais problemas de suporte analisados.

Atividade de suporte para	FlexLM	ÉlanLM	iFOR/LS
Re-inicialização manual do servidor de licenças	SIM	NÃO	SIM
Manipulação do arquivo de licenças	SIM	SIM	SIM
Configuração da localização dos servidores replicados	SIM	SIM	SIM
Liberação de licenças pendentes manualmente	SIM	NÃO	SIM
Limitação no número de réplicas do servidor	SIM	NÃO	NÃO
Interação com desenvolvedores para cálculo de chaves de licenças e/ou ativação do módulo servidor	SIM	SIM	SIM

Tabela 2.5 - Principais problemas de suporte encontrados nos gerenciadores de licença.

Embora o serviço de licenciamento seja uma boa opção e venha sendo adotado por um número cada vez maior de produtores/vendedores de software, o quadro acima mostra alguns de seus problemas. Nossa proposta é projetar um serviço de licenciamento que elimine os problemas ocasionados pelo considerável incremento e dependência das atividades de suporte levantadas acima, tanto no ambiente do desenvolvedor quanto do comprador. Além disso, este serviço deverá apresentar um baixo custo de produção e aquisição, sendo uma alternativa apropriada para que produtores de software de prateleira incorporem proteção contra uso ilegal aos seus produtos sem que isto os torne mais caros. O próximo capítulo é destinado a apresentar esta proposta.

3

3 O Bouncer

Neste capítulo será apresentado um serviço de licenciamento denominado Bouncer, incluindo seus objetivos, justificativa, benefícios, arquitetura e protocolo de licenciamento. No capítulo seguinte discutiremos a infra-estrutura de comunicação necessária para sua implementação.

O Bouncer é um sistema distribuído que permite controlar o uso de licenças de software em ambientes de redes locais, oferecendo uma API a ser usada pelas aplicações que irão se beneficiar de seus serviços.

Como se pode observar no capítulo 2, que trata do serviço de licenciamento, este serviço oferece uma grande flexibilidade funcional e administrativa. Através da combinação de vários atributos, políticas de licenciamento sofisticadas podem ser definidas para os produtos de software. Além disso, conversão de cópias de demonstração de produtos em cópias finais, alteração do número de licenças disponíveis, liberação da execução de uma aplicação em nós da rede ou para usuários específicos, tudo isto pode ser feito com a simples reconfiguração da política de licenciamento destes, através do gerenciador de licenças. Em contrapartida, alguns problemas que geram suporte administrativo foram observados neste tipo de ferramenta.

Diferentemente das outras soluções de gerenciamento de licenças, vistos no capítulo anterior, que utilizam o modelo de programação cliente-servidor, o Bouncer adota um modelo híbrido de programação distribuída, mesclando os paradigmas *peer-to-peer* (P-P) e cliente-servidor (C-S). O modelo de programação C-S tem como base a comunicação entre um elemento que solicita a resolução de um determinado problema (o cliente) a um outro elemento, obtendo deste (o servidor) os resultados de que necessita. Este tipo de comunicação envolve um protocolo bastante simples, do tipo pedido-resposta, onde a resposta do servidor ao pedido do cliente funciona como uma confirmação do pedido enviado. Mais detalhes sobre o modelo C-S serão dados no próximo capítulo. No modelo *P-P*, uma mesma

entidade acumula as funcionalidades de cliente e de servidor, hora encaminhando pedidos de prestação de serviços a entidades capazes de prestá-los, hora provendo serviços para outras entidades.

Como deve ser característica obrigatória em todas as ferramentas que se propõem a controlar utilização de software, o Bouncer procura ser o mais robusto e tolerante a faltas possível. Detalhes de como isto é garantido serão vistos mais adiante. Porém, o principal diferencial a ser imposto pelo Bouncer é a eliminação dos problemas de suporte e administração do serviço de licenciamento.

O objetivo do Bouncer é o de proteção, ou seja, permitir que somente um determinado número de usuários licenciados executem e utilizem uma determinada aplicação simultaneamente. A principal política de licenciamento adotada pelo Bouncer é a de **quantidade de ativações concorrentes** ou número máximo de licenças ativas simultaneamente, visto que esta é a política mais popular em ambientes distribuídos. Porém, com uma certa manipulação de parâmetros especiais da API, exemplificada oportunamente, também é possível implementar as políticas básicas **conjunto de compartilhamento** e **anotações de licenças**. A redução de flexibilidade do Bouncer com relação a outros gerenciadores de licenças trás como contrapartida a eliminação das atividades de suporte administrativo com o serviço de licenciamento. Desta forma, o Bouncer se mostra uma alternativa de serviço de licenciamento simples, criativa e *plug-and-play*, podendo ser adquirido por desenvolvedores por um custo mais baixo. Todas estas características o tornam, no nosso entender, uma solução mais adequada para desenvolvedores de software de prateleira, cuja proposta é a de ter um baixo custo de desenvolvimento e, conseqüentemente, um preço acessível para o usuário final.

O Bouncer oferece vantagens tanto para desenvolvedores de software quanto para administradores. Como vantagens para os desenvolvedores, podemos apontar redução do esforço de programação no desenvolvimento de aplicações que possuem requisitos de licenciamento, flexibilidade para implementação da política de licenciamento a nível de programação e flexibilidade para a construção de ferramentas de monitoração para o administrador do sistema/rede.

Do ponto de vista do administrador do sistema/rede, se comparado a outros serviços de licenciamento existentes, o Bouncer, uma vez instalado, dispensa qualquer atenção por parte do administrador, mesmo no caso de falha de algumas máquinas da rede ou de algum dos seus processos. Além disso, uma alta disponibilidade do serviço é garantida de forma trivial, pela sua arquitetura e seu protocolo.

Como a proposta do Bouncer é a de ser um gerenciador de licenças simples, algumas funcionalidades oferecidas por outros gerenciadores de licença mais sofisticados, caso desejadas, deverão ser implementadas pelo desenvolvedor.

- O Bouncer não permite a conversão de cópias de demonstração em cópias definitivas. Esta, porém, não é uma operação comum no

segmento de mercado de software que o Bouncer se propõe a atender. Além do mais, não é interessante para pequenos desenvolvedores manter uma infra-estrutura para prestar este tipo de suporte.

- O número máximo de licenças permitido a uma aplicação é passado diretamente como parâmetro para a API. Qualquer modificação neste parâmetro deverá ser implementada pelo desenvolvedor, através de algum esquema de personalização de produtos por ele utilizado. Este é o tipo de operação que será realizada por desenvolvedores de software de prateleira antes da comercialização do produto. Por exemplo: O software Alfa é disponibilizado nas lojas nas versões para 10, 25 e 50 usuários. Não é comum que clientes de produtos de software desta categoria queiram fazer um *upgrade* a partir de uma cópia já em uso. O esquema de personalização, portanto, não causará nenhum desconforto para o desenvolvedor.
- Outras modalidades de licenciamento não cobertas pelo Bouncer, caso desejadas, deverão ser implementadas pelo desenvolvedor. Isto, no entanto, não se configura como uma limitação grave para o Bouncer, pois em ambientes de rede, a modalidade de licenças mais popular é a de quantidade de ativações concorrentes, principalmente quando se trata de software de prateleira, onde normalmente não são utilizadas políticas de licenciamento mais sofisticadas. Uma modalidade de licenciamento bem simples de ser implementada pelo desenvolvedor é a de data de expiração, através da simples inserção de um teste de uma data previamente personalizada (*hardwired*) com a data do sistema no início do código da aplicação. Após expirada esta data, a aplicação não mais funcionará.

3.1 Arquitetura e Serviços

O serviço de licenciamento é composto por dois módulos: um com a funcionalidade de cliente e outro com a de servidor. De forma simplificada, o cliente é responsável pela solicitação de pedidos de licenças para a execução de aplicações. Já o servidor gerencia as licenças solicitadas pelos clientes. As aplicações possuem a funcionalidade cliente incorporadas ao seu código executável, através da link-edição das funções da API do Bouncer.

A funcionalidade servidora se encontra isolada em um módulo executável a parte que tornar-se-á um *daemon*. Cada máquina da rede deverá possuir um, e somente um, servidor ativo. Este fato será detalhado mais adiante.

Todas as operações realizadas pelos servidores são coordenadas por um protocolo, denominado protocolo Bouncer, que trata desde a inicialização de

servidores por parte das aplicações clientes até auditoria de licenças, tanto a nível local quanto distribuído.

O serviço de licenciamento Bouncer adota um modelo híbrido, utilizando de forma conjunta os paradigmas C-S e P-P. Desta forma, temos a simplicidade do modelo C-S associada à maior transparência e disponibilidade do modelo P-P. Do ponto de vista de uma única máquina, temos o modelo C-S, onde existe um servidor dedicado e bem conhecido, enquanto que, do ponto de vista da rede como um todo, temos uma arquitetura P-P, onde todos os nós são iguais e trocam informações entre si.

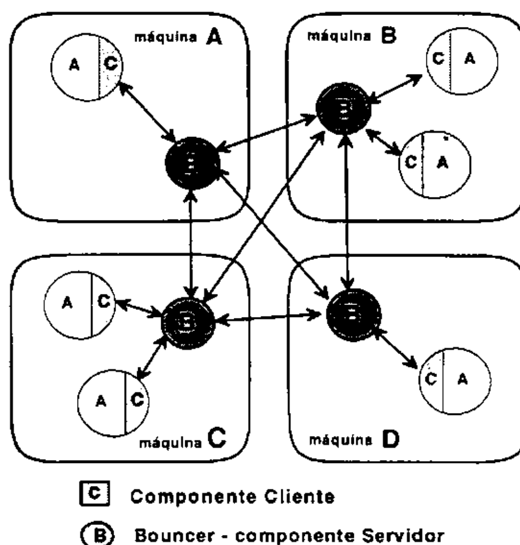


Figura 3.1 - Modelo Híbrido do Serviço Bouncer

O processo de solicitação de licenças (comunicação entre componente cliente e servidor) se dá conforme mostrado na figura 2.1 do capítulo anterior. Na figura 3.1, é dada uma visão global do funcionamento do Bouncer em ambiente distribuído. Apenas um *daemon* servidor de licenças, denominado de processo B, deve estar ativo em cada máquina onde há pelo menos uma aplicação que use o serviço Bouncer. Cada processo B recebe pedidos de licença das aplicações rodando em sua respectiva máquina.

O procedimento de obtenção de licença inicia-se quando uma aplicação cliente comunica-se diretamente com o processo B de sua máquina, solicitando deste uma licença para que possa continuar sua execução. O processo B, então, consulta uma base de dados mantida por ele localmente para verificar quantas cópias da aplicação que solicitou a licença estão em execução em toda a rede. Esta informação está disponível localmente porque cada servidor B presente na rede envia informações sobre as licenças utilizadas em sua máquina para os outros servidores da rede. Desta forma, todos os processos B possuem localmente uma base de dados com um panorama global de utilização das licenças, e estas bases são todas consistentes entre sí. Logo, através da cunsulta a esta base de dados, o

servidor interpelado pela aplicação cliente tem subsídios para decidir se concede, ou não, a licença para esta aplicação. Na seção 3.3 descreveremos em detalhes o protocolo Bouncer executado pelos servidores B.

3.2 Interface de Programação

A API do Bouncer oferece um conjunto padronizado de funções para a linguagem C. Esta linguagem foi escolhida pelas seguintes razões: 1) é uma linguagem amplamente difundida entre os desenvolvedores, além de ser compilável com a utilização de ferramentas de desenvolvimento para C++, e 2) é praticamente um padrão de mercado em desenvolvimento de software básico. É bom lembrar que não existe nenhuma restrição quanto ao uso de outra linguagem de programação para a implementação da API, desde que esta seja adequada ao desenvolvimento de software básico. As funções da API permitem ao programador integrar às aplicações funcionalidade de licenciamento e proteção em redes locais. A API é composta por quatro funções, descritas na tabela abaixo.

Função	Descrição
B_Request()	Solicita ao Bouncer uma licença para executar uma aplicação. Esta função bloqueia a aplicação do desenvolvedor até que o resultado do pedido de licença seja retornado.
B_Release()	A aplicação do desenvolvedor que ativar esta função estará liberando a licença para que esta possa ser utilizada por outra aplicação.
B_Monit()	Permite consultar o Bouncer e obter um conjunto de informações sobre as aplicações protegidas: host em que executam, nome, número total de licenças disponíveis, número de licenças consumidas (em uso), etc.
B_Check()	Monitora a existência do Bouncer, informando se este está ativo ou se, por algum motivo, foi desativado.

Tabela 3.1 - Funções disponíveis na API do Bouncer.

3.2.1 B_Request

A função B_Request solicita uma licença de execução. Esta função é responsável pela inicialização do serviço de licenciamento, que consiste na criação do processo Bouncer, responsável pela concessão de licenças de execução das aplicações. Portanto, antes de qualquer coisa, a componente cliente da aplicação irá tentar inicializar o *daemon* Bouncer.

Para sincronizar o processo de inicialização do *daemon* Bouncer, evitando que mais de um cliente o faça simultaneamente, uma região crítica mutuamente exclusiva é travada através de um semáforo, indicando que já existe uma aplicação tentando inicializar o serviço. Uma vez habilitado a tentar ativar o servidor, o cliente tenta acessar uma área compartilhada mutuamente exclusiva, área esta

permanentemente travada pelo *daemon* Bouncer enquanto este estiver ativo. Caso a aplicação consiga acessar a área compartilhada, o que indica que o servidor não está ativo, ela deverá inicializar o servidor Bouncer. Após a verificação da existência ou não do servidor Bouncer, a componente cliente deverá prosseguir com a solicitação da licença de execução.

A solicitação de licença normalmente é feita no início do código do desenvolvedor. O resultado do pedido de licença é retornado em um dos parâmetros da função, como descrito adiante.

Formato

Status = **B_Request**([in] *ProdtId*, [in] *Verifier*, [in] *MaxLicenses*, [out] *Result*);

char*	*ProdtId*;
char*	*Verifier*;
int	*MaxLicenses*;
int*	*Result*;
int	*Status*;

Argumentos

ProdtId - *String* que identifica o produto a ser protegido, podendo ser construído da seguinte forma: **nome + versão + número de série**. Esta identificação está a cargo do desenvolvedor.

Verifier - Parâmetro que identifica a instância de execução da aplicação em um determinado nó da rede. Seu preenchimento é opcional, sendo utilizado como complemento da identificação da aplicação. *Verifier* é utilizado para permitir a implementação de licenças compartilhadas. Assim, em um ambiente gráfico baseado em janelas, várias aplicações (uma em cada janela) podem vir a consumir apenas uma licença. Para a obtenção deste efeito, basta que os dados que irão compor este campo sejam comuns a todas as instâncias da aplicação (identificador do líder de grupo de processos + endereço IP do host, por exemplo).

MaxLicenses - Indica o número de licenças de execução permitidas para o produto em questão, isto é, o número máximo de cópias de aplicações que podem ser executadas simultaneamente.

Result - Resultado do pedido de licença. Assume o valor das constantes **B_LICENSE_OK** ou **B_NO_LICENSE.**

Status - B_Request( ) retornará 0, caso obtenha sucesso ou um código de erro a ser tratado pelo desenvolvedor. Veja alguns exemplos de possíveis códigos de erro retornados pelas funções da API do Bouncer na tabela 3.2.

Códito do erro	Descrição
E_INITDAEMON	erro na inicialização do *daemon* do Bouncer.
E_DTIMEOUT	erro de *timeout* na criação do *daemon* do Bouncer.
E_INSERTTABLE	erro de inserção na tabela de licenças.
E_DELETE	erro na remoção de registro na tabela de licencas.
E_NODAEMON	*daemon* não está em execução.
E_IPHOST	erro na obtenção do endereço IP do *host* local.
E_AUTHFAIL	falha de autenticação.
E_GETLIST	erro na montagem da lista do monitor.
E_NOPRODT	nenhum identificador de produto foi passado.

Tabela 3.2 - Alguns códigos de erro retornados pela API do Bouncer.

Exemplo

Através de um trecho de código escrito em linguagem C, o exemplo mostra como é possível incorporar de forma simples as facilidades do Bouncer às aplicações desenvolvidas.

```
#include <stdio.h>
#include <bapi.h>       /* descricao da interface de programacao do Bouncer */
#include <berror.h>     /* definicao dos codigos de erro retornados pela API do Bouncer */

/*      Ponto de entrada
*/
main(){
int     licenca;

/*      Solicitacao da licenca
*/
        if (B_Request("FGENER 1.3", "", 10, &licenca) != 0 ){
                fprintf( stderr, "Erro na chamada a  B_Request\n");
                /* Neste ponto, o desenvolvedor decide que atitude tomar */
        }
        if ( licenca == B_NO_LICENSE ){
                fprintf( stderr, "Licencas esgotadas para esta aplicação\n");
                /* Neste ponto, o desenvolvedor decide que atitude tomar */
        }
        ... corpo da aplicação ...
}
```

3.2.2 B_Release

A partir do momento em que for chamada, B_Release liberará a licença consumida pela respectiva aplicação para a base de licenças disponíveis. Ao ser ativada para liberar a última licença controlada por um determinado *daemon* Bouncer, B_Release irá causar o fim da execução de tal *daemon*. É importante observar que este processo de finalização do *daemon* servidor pode gerar uma condição de corrida, se realizado simultaneamente a uma solicitação de licença feita através da função B_Request. Como visto anteriormente, B_Request verifica a existência do *daemon* servidor B antes de solicitar-lhe uma licença tentando acessar uma área mutuamente exclusiva travada pelo *daemon* servidor. Pode ocorrer que uma aplicação cliente que ativou B_Request constate que B está ativo (região mutuamente exclusiva travada por B), não tentando, portanto, re-inicializá-lo, e perca a CPU antes que a requisição de licença seja enviada. Neste meio tempo, suponha que B esteja finalizando sua execução devido à liberação da última licença por ele controlada. Quando a aplicação cliente recuperar a CPU, não mais haverá um servidor ativo para atender ao pedido de licença. Isto pode ser resolvido através da determinação de uma região crítica mutuamente exclusiva, protegida por semáforos, que garanta atomicidade nas transações de inicialização e finalização do *daemon* B.

Formato

Status = **B_Release**([in] *ProdtId,* [in] *Verifier*);

Argumento

ProdtId - *String* que identifica o produto cuja licença será liberada.

Verifier - No caso do uso de licenças compartilhadas, este parâmetro é necessário para indicar que instância da aplicação está devolvendo sua licença. Se este parâmetro não foi utilizado na solicitação da licença (licença não é compartilhada), o seu valor pode ser uma *string* vazia.

Status - B_Release( ) retornará 0, caso obtenha sucesso ou um código de erro a ser tratado pelo desenvolvedor.

3.2.3 B_Monit

Permite ao desenvolvedor construir ferramentas para monitorar aplicações protegidas, tanto a nível local quanto remoto. Esta função obtém informações do Bouncer sobre as aplicações por ele protegidas. Com esta função é possível levantar estatísticas sobre a utilização das aplicações no ambiente do cliente. Além do mais, por ser possível monitorar máquinas específicas da rede, através de ferramentas desenvolvidas utilizando a função B_Monit, o administrador poderá detectar eventuais segmentações na sua rede local.

Formato

Result = **B_Monit**([in] *ProdtId*, [in] *Host*, [out] *VetLen*);

```
char*          ProdTId;
char*          Host;
int*           VetLen;
struct Node*   Result;

struct Node {
        char                IPAddr[ B_MAXIPLEN +1];
        int                 Error;
        int                 VetLen;
        struct Prodtnode    VetProd[ B_MAXPRODNODES +1];
};

struct Prodtnode {
        char     ProdtId[B_MAXPRODTLEN + 1];
        int      MaxLicenses;
        int      ActLicenses;
};
```

Argumentos

ProdtId - *String* que identifica a aplicação a ser monitorada. Caso não seja informado (*string* vazia), todas as aplicações controladas pelo Bouncer serão monitoradas.

Host - Indica a máquina que será monitorada pelo Bouncer. Caso não seja informado (string vazia), todas as máquinas da rede serão monitoradas.

VetLen - Número de posições do vetor de respostas. Este número indica a quantidade de processos B que efetivamente responderam à solicitação de monitoração (caso em que várias máquinas foram interpeladas).

Result - B_Monit( ) retornará o apontador para um vetor de estruturas do tipo Node, caso obtenha sucesso, ou o apontador nulo (NULL), caso contrário. Para cada máquina conectada à rede que respondeu ao pedido de monitoração, existirá sua respectiva estrutura Node, contendo seu endereço IP e um vetor com as informações solicitadas. Estas informações serão tratadas pelo desenvolvedor como bem lhe convier.

Veja a descrição dos campos da estrutura Node.

IPAddr - Endereço IP (notação **dot**) da máquina origem dos dados. No caso da não existência de rede, o campo *IPAddr* virá vazio.

Error - Código de erro ocorrido durante a monitoração.

VetLen - Tamanho do vetor de estruturas do tipo Prodtnode. Indica a quantidade de aplicações distintas em execução na máquina monitorada.

VetProd - Vetor de estruturas Prodtnode. Estas estruturas trazem informações sobre cada aplicação, conforme definição abaixo.

A estrutura Prodtnode é composta pelos seguintes campos:

ProdtId - Identificador do produto.

MaxLicenses - Número máximo de licenças permitidas para a aplicação.

ActLicenses - Número de licenças ativas da aplicação. Este número corresponde ao número de cópias da aplicação executando no momento da monitoração.

Exemplo

Para demonstrar o uso da função B_Monit(), foi implementado o comando de linha **monitor**, cujo código fonte, escrito em linguagem C, está listado no apêndice deste trabalho. A seguir, descreveremos a sintaxe do comando monitor e alguns exemplos de sua utilização.

Sintaxe

monitor [-a<prodtid>] [-h<hostname>]

Descrição dos parâmetros

prodtid - String identificador do produto cujas licenças irão ser monitoradas.

hostname - Nome da máquina a ser monitorada.

Digamos que o administrador da rede queira monitorar as cópias da aplicação protegida Alpha na máquina nephastus.paqtc.rpp.br. O monitor será chamado da seguinte forma:

$ monitor -aAlpha -hnephastus.paqtc.rpp.br

O resultado exibido poderá ser, por exemplo:

```
                         MONITOR
*****************************************************************
Host: nephastus.paqtc.rpp.br.
```

Produto	Maximo de Copias	Copias Ativas
Alpha	5	2

Caso o parâmetro *prodtid* seja omitido, todas as aplicações protegidas em execução na máquina especificada serão monitoradas.

M O N I T O R

****

Host: nephastus.paqtc.rpp.br.

Produto	Maximo de Copias	Copias Ativas
Alpha	5	2
lanwatcher	3	1
netacme	7	1

Na ausência do parâmetro *hostname*, todas as máquinas da rede local serão monitoradas. Portanto, para cada uma delas, o grupo de informações apresentado acima será mostrado.

M O N I T O R

****

Host: nephastus.paqtc.rpp.br.

Produto	Maximo de Copias	Copias Ativas
Alpha	5	2
lanwatcher	3	1
netacme	7	1

Host: piazzola.paqtc.rpp.br.

Produto	Maximo de Copias	Copias Ativas
editor	10	2
lanwatcher	3	2
netacme	7	2

Host: amadeus.paqtc.rpp.br.

Produto	Maximo de Copias	Copias Ativas
Alpha	5	1
netacme	7	3

O administrador pode, inclusive, escrever scripts que periodicamente executem o monitor, armazenando seus resultados em um arquivo. Este arquivo, posteriormente, poderá servir como entrada para um programa que gere estatísticas de utilização das aplicações. Além disso, como mencionado antes, através do monitor, o administrador poderá detectar particionamento na rede, uma vez que algumas máquinas podem passar a não constar no relatório de monitoração.

3.2.4 B_Check

Esta função não recebe nenhum argumento, apenas testa se o processo Bouncer continua ativo. B_Check retorna 0 quando encontra o servidor ativo ou um código do erro a ser tratado pelo desenvolvedor, caso contrário. B_Check é especialmente importante para recuperar o serviço de licenciamento de eventuais falhas, como será detalhado adiante.

B_Check( ) pode ser ativada em qualquer parte do código do desenvolvedor de aplicações, ficando a cargo deste implementar o esquema de verificação que lhe pareça melhor.

Formato

Status = **B_Check**( );

Argumentos

Status - B_Check retornará o valor 0 caso detecte que o servidor Bouncer esteja ativo. Caso contrário, um código de erro é retornado (ver tabela 3.2).

Exemplo

Melhorando o exemplo dado para ilustrar a utilização da função B_Request, vamos implementar uma verificação periódica da existência do *daemon* Bouncer.

```
#include <signal.h>

main(){
        ... após licença ter sido obtida ...
        /*
         * Configura o alarme do UNIX para verificar periodicamente a cada quinze segundos
         * a existencia do daemon Bouncer através de uma função que utiliza B_Check()
         */
        alarm( 15 );
        signal( SIGALRM, CheckB );
                ...
```

```
}

/* CheckB
 * objetivo:
 *      Verificar a existencia do servico de licenciamento utilizando B_Check()
 */
void
CheckB(){
        int     licenca;

        if( B_Check() != 0 ){
                fprintf( stderr,"Bouncer desativado\n");
                /* uma nova licenca eh solicitada para recuperar o Bouncer. */
                if( B_Request("FGENER 1.3", "", &licenca ) != 0 )
                        fprintf( stderr, "Erro na chamada a  B_Request\n");
                        /* Neste ponto, o desenvolvedor decide que atitude tomar */
                }
                if ( licenca == B_NO_LICENSE ){
                        fprintf( stderr, "Licencas esgotadas para esta aplicação\n");
                        /* Neste ponto, o desenvolvedor decide que atitude tomar */
                }
        }
        alarm( 15 );
        signal( SIGALRM, CheckB );
        return;
}
```

3.3 O protocolo Bouncer

Passaremos agora a descrever o protocolo Bouncer, que rege todo o funcionamento do gerenciamento de licenças.

3.3.1 O Protocolo Bouncer em um Ambiente Livre de Falhas

Para simplificar a apresentação, inicialmente vamos descrever o funcionamento do protocolo Bouncer assumindo que nenhum tipo de anormalidade irá ocorrer (e.g. falhas em máquinas, em processos ou qualquer tentativa de violação).

Além de um ambiente livre de falhas, iremos também assumir alguns pressupostos:

P1. Sempre que existir pelo menos uma aplicação cliente em execução em uma máquina, um, e apenas um servidor B executando o protocolo Bouncer deverá estar ativo naquela máquina.

P2. Antes de finalizar sua execução, a aplicação libera a licença que detém.

Na próxima seção mostraremos como estes pressupostos podem ser garantidos pelo módulo cliente do Bouncer. Também serão apresentadas as modificações necessárias para que o Bouncer ofereça seu serviço mesmo na presença de falhas.

O protocolo Bouncer é executado por um grupo formado pelos processos B, que desempenham a funcionalidade de servidores de licença. Cada um dos processos B deve estar executando em máquinas da rede onde exista pelo menos uma aplicação protegida sendo executada. Daqui por diante, passaremos a tratar este grupo como **grupo Bouncer**.

Vamos assumir que existem mecanismos para gerenciar o grupo Bouncer. Estes mecanismos devem permitir que o grupo possa ser criado, dissolvido, receber e liberar membros. Cada grupo possui um processo com o *status* especial de **líder de grupo**. O processo líder de um grupo deve ser o seu membro mais antigo. Detalhes de como e quando um processo B é inicializado e se junta ao grupo Bouncer serão vistos mais adiante.

Os processos que compõem o grupo Bouncer trabalham de forma cooperativa para controlar as licenças em um ambiente distribuído. Para que isto seja possível, é necessário que estes processos troquem informações entre si. A forma utilizada para a comunicação entre os processos que compõem o grupo Bouncer possui algumas características especiais:

1. Um processo B membro do grupo Bouncer pode enviar mensagens endereçadas ao grupo, que serão recebidas por todos os seus membros, inclusive pelo processo que a enviou.
2. Caso uma mensagem enviada para o grupo Bouncer não possa ser entregue a todos os seus membros, nenhum deles a receberá.
3. Quando várias mensagens são enviadas para o grupo Bouncer, seja por um mesmo processo B, seja por processos distintos, todos os membros do grupo recebem-nas em uma mesma ordem. Por exemplo, o processo B1 envia as mensagens M1, M2 e M3 ao grupo Bouncer. Se um dos processos do grupo receber estas mensagens na ordem M3, M1, M2, garantidamente o restante do grupo irá recebê-las nesta mesma ordem.

No próximo capítulo, que trata de infra-estrutura de comunicação entre processos no Bouncer, definiremos com mais detalhes esta forma de comunicação entre processos.

A figura 3.2 apresenta o protocolo que rege o funcionamento dos servidores Bouncer, utilizando, para isto, uma notação baseada na linguagem concorrente CSP (*Communicating Sequential Processes*) [Hoare 78].

```
1  BOUNCER =
2    Client(i, j) ? License_j →
3    [ NumLicenses_j = MaxLicenses_j → Client(i, j) ! NoLicense();
4    [] NumLicenses_j < MaxLicenses_j → Bouncer (i:1..n) ! License_j ;
5    ]
6    ||
7    Bouncer (i:1..n) ? License_j →
8    [ NumLicenses_j < MaxLicenses_j →
9       RECORD_LICENSE(j, TabLicenses);
10       [ MyClient(i) →
11          Client(i, j) ! LicenseOk();
12       ]
13    [] NumLicenses_j = MaxLicenses_j →
14       [ MyClient(i) →
15          Client(i, j) ! NoLicense();
16       ]
17    ]
18    ||
19    Client(i,j) ? Release_j → Bouncer (i:1..n) ! Release_j;
20    ||
21    Bouncer (i:1..n) ? Release_j → Cliente(i, j) ! RELEASE_LICENSE(j, TabLicenses);
22    ||
23    Client(i,j) ? Monitore_j → Client(i,j) ! MONITOR(j);
```

Figura 3.2 - O protocolo Bouncer.

Ao receber um pedido de licença de execução de um cliente (linhas de 2 a 5 da figura 3.2), B irá verificar a disponibilidade de licenças para aquela aplicação em sua tabela de licenças. Caso não haja nenhuma licença disponível, B responderá negativamente ao cliente. Caso contrário, o pedido de licença será repassado para o grupo Bouncer. Os pedidos de licença só irão ser efetivamente cadastrados quando a mensagem for recebida do grupo, pois este pedido estará concorrendo com eventuais pedidos feitos por outros membros do grupo. Como as mensagens serão entregues ordenadas ao grupo, não haverá inconsistências.

Ao receber um pedido de licença do grupo (linhas de 7 a 17 da figura 3.2), B fará uma busca por licenças disponíveis para aquela aplicação em sua tabela. Havendo licença disponível, esta será cadastrada na tabela, dando-lhe o *status* de "em uso". O B que enviou o pedido ao grupo irá, então, informar ao seu cliente que a licença foi concedida e que a aplicação pode prosseguir com sua execução. Se não houver mais nenhuma licença disponível, este mesmo B negará o pedido feito por seu cliente.

De acordo com a linha 19 da figura 3.2, quando uma liberação de licença é recebida de um cliente, B a repassa para o grupo. Só as liberações recebidas do grupo serão processadas pelo B que enviou a mensagem. Isto garante que todo o grupo também recebe a mensagem e, portanto, as tabelas são mantidas consistentes. Na linha 21, ao receber uma liberação de licença do grupo, aí sim, B remove a entrada de sua tabela referente a esta licença.

Caso receba um pedido de monitoração (linha 23), B devolverá ao cliente uma estrutura de dados preenchida com o conteúdo de sua tabela de licenças.

O símbolo '||' indica que os blocos de comandos por ele separados podem ser executados concorrentemente. Desta forma, enquanto B está processando uma solicitação de licença para uma aplicação, este também está apto a receber um pedido de monitoração ou uma liberação de licenças vindo do grupo, de maneira concorrente.

3.3.2 O Protocolo Bouncer em um Ambiente Sujeito a Falhas

Agora consideraremos o protocolo Bouncer sendo executado em um ambiente onde falhas podem ocorrer. Isto irá requerer que o protocolo Bouncer sofra uma série de adequações que o torne imune a tais falhas. Além disso, será mostrado como os clientes do Bouncer procedem para validar os pressupostos assumidos para o bom funcionamento do protocolo Bouncer, citados na seção anterior.

3.3.2.1 Validação dos Pressupostos

Nesta seção discutiremos como a componente cliente do Bouncer, incorporada às aplicações do desenvolvedor, garante a validade dos pressupostos apresentados na seção 3.3.1, que são imprescindíveis para o correto funcionamento do protocolo Bouncer.

P1. *"Sempre que existir pelo menos uma aplicação cliente em execução em uma máquina, um, e apenas um servidor B executando o protocolo Bouncer deverá estar ativo naquela máquina."*

Quando uma aplicação integrada ao serviço de licenciamento Bouncer é inicializada por um usuário, esta pede uma licença de execução ao servidor através da função B_Request(), pertencente à API Bouncer. Está embutida nesta função a verificação da existência de um servidor Bouncer (B) em execução. Caso B não esteja ativo, a aplicação irá ativá-lo. Neste processo está incluído um mecanismo de exclusão mútua, caso mais de um cliente tente inicializar B simultaneamente (situação em que dois usuários inicializaram uma aplicação protegida mais ou menos no mesmo instante).

No momento em que um servidor B entra em execução, sua primeira atitude é procurar juntar-se ao grupo Bouncer, formado por outros servidores B. Caso este

grupo não exista, isto significa que este é o único B em execução na rede naquele instante e, portanto, deverá criar o grupo de processos Bouncer.

Uma vez engajado ao grupo Bouncer, o servidor B deverá obter o estado do grupo, ou seja, a tabela contendo informações sobre todas as licenças que estão sendo utilizadas naquele momento. Este estado será solicitado ao B líder do grupo. Após cumpridas estas etapas iniciais, B estará apto a atender seus clientes.

Como se trata de um ambiente sujeito a faltas, é possível que o processo B apresente falhas, seja porque a máquina inteira falhou, seja porque o processo foi desativado por algum motivo (por exemplo, o administrador ou algum usuário com certos privilégios cancelou sua execução).

Quando a máquina falha, tanto o servidor B quanto as aplicações clientes irão ser cancelados. Quando a máquina voltar a funcionar, não existirá nenhuma aplicação cliente em execução e, portanto, não deve haver nenhum servidor B executando (nenhum cliente o inicializou ainda).

Quando o processo B falha e as aplicações clientes permanecem ativas, cabe a estas aplicações detectar a falha do servidor e providenciar sua re-inicialização. Para resolver este problema, periodicamente, o cliente irá fazer uma verificação da existência do servidor. Isto é feito através o uso da função B_Check(), pertencente à API do Bouncer. A eventual resposta do servidor deverá ser autenticada pelo cliente, para que respostas de falsos servidores sejam descartadas. Caso seja constatado que o servidor morreu, o cliente irá solicitar nova licença de execução. Neste processo, como já visto anteriormente, está envolvida a re-inicialização de B, incluindo toda a sincronização e exclusão mútua, caso mais de um cliente tente fazer isto simultaneamente. Se a licença não for concedida, a aplicação cliente procederá conforme a política adotada pelo desenvolvedor.

P2. *"Antes de finalizar sua execução, a aplicação libera a licença que detém."*

A aplicação do desenvolvedor deve ser projetada de forma a liberar a licença que consome antes do final de sua execução. Isto deverá ser feito para que as licenças possam ser disponibilizadas para outras cópias desta aplicação que venham a ser executadas no futuro. Como visto na especificação da interface de programação do Bouncer, a função a ser utilizada para liberar uma licença é B_Release(). Portanto, para garantir a validade deste pressuposto, basta que o desenvolvedor inclua uma chamada a B_Release() antes de finalizar a aplicação.

Como pode ser observado, a validade dos pressupostos pode facilmente ser garantida pela funcionalidade cliente do Bouncer, incorporada às aplicações.

3.3.2.2 Análise dos Possíveis Problemas e Soluções

Uma série de situações indesejáveis podem acontecer, sejam elas pura fatalidade ou verdadeiras sabotagens. Aqui levantamos uma série de situações que podem prejudicar o bom funcionamento do protocolo Bouncer. São elas:

S1 - O servidor B falha.

Isto pode ocorrer por dois motivos: o processo servidor falhou ou a máquina onde o servidor executava falhou.

A recuperação do Bouncer se dará da seguinte forma: quando apenas o processo servidor falhou, isto será detectado por suas aplicações clientes através do uso da função B_Check. Neste caso, o servidor será re-inicializado por uma das aplicações clientes (veja pressuposto P1). Caso a falha tenha sido na máquina onde o servidor executava, este será re-inicializado pela primeira aplicação que for executada por algum usuário (veja pressuposto P1).

Do ponto de vista dos servidores que executam em outras máquinas da rede (o restante do grupo Bouncer), podem ocorrer duas situações distintas: i) o grupo detectou a falha do servidor B; ii) o servidor B se recuperou antes que o grupo detectasse sua falha.

No caso i), quando é notada a perda de um membro do grupo, este deve ser refeito, sendo excluído o membro ausente. Além disso, todas as licenças alocadas para a máquina do servidor que falhou deverão ser excluídas das bases de licenças de todo o grupo.

Como visto neste capítulo, ao ser re-inicializado, B irá recuperar o estado do grupo, através da tabela de licenças enviada pelo líder do grupo. Na circunstância descrita em ii), pode acontecer de haver na tabela de licenças recém recebida por B, licenças oriundas de sua própria máquina. É necessário, portanto, fazer uma auditoria local para detectar quais destas licenças ainda são válidas, propagando para o grupo as eventuais alterações. Note que a situação ii) não é freqüente, pois o grupo deverá ser ágil em detectar falhas em seus membros.

S2 - Uma aplicação cliente tem sua execução encerrada sem liberar a licença que consumia.

O pressuposto P2 garante que, em condições normais, isto não irá acontecer. Estamos supondo, porém, que uma aplicação pode ter sua execução encerrada de forma anormal. Nestas circunstâncias, nenhuma atitude imediata será tomada pelo Bouncer. A partir do momento em que as licenças para uma determinada aplicação se esgotarem, ao invés de negar a licença para o usuário que deseja executá-la, o Bouncer envia uma solicitação de auditoria para o grupo de servidores. Neste momento, cada servidor irá varrer sua tabela de licenças, procurando por aplicações protegidas que estejam em execução em sua máquina. O Bouncer testará se cada ocorrência localizada ainda está ativa. Para cada eventual

aplicação inválida da tabela, será enviado um pedido de liberação de sua licença para o grupo. A requisição de licença ficará pendente até que o processo de auditoria seja concluído. Após a conclusão da auditoria, o servidor B poderá responder ao cliente, certo de que sua resposta está embasada em dados atualizados e fidedignos. O procedimento de auditoria será detalhado adiante, na seção 3.3.3.

S3 - Foi instalado um B falso, que tanto pode mandar falsas mensagens para o grupo quanto responder o que bem entender aos clientes (liberar licenças para todos ou negá-las sempre que receber um pedido).

Todas as mensagens trocadas, tanto entre os membros do grupo Bouncer quanto entre as aplicações clientes e o servidor B deverão ser autenticadas. Esta autenticação consiste na utilização de criptografia de chave única para criptografar as mensagens trocadas entre os processos do serviço Bouncer. O processo origem da mensagem irá criptografá-la utilizando uma chave que também será utilizada pelo processo destino para decriptografá-la. Caso o destino não consiga descriptografar a mensagem, isto indica que esta foi enviada por um processo impostor. Em [Schneier 93] é apresentado um algoritmo para esta finalidade.

S4 - A rede foi particionada em várias sub-redes. Para cada sub-rede irá surgir um sub-grupo de processos, uma vez que, sob o ponto de vista dos servidores de um dos segmentos isolados, os servidores dos outros segmentos falharam. Cada grupo irá disponibilizar, então, o mesmo número de licenças que havia para o grupo original inteiro (o número de licenças foi multiplicado pelo número de sub-redes).

O problema do particionamento da rede é de difícil solução. Vincular a execução dos servidores a um determinado domínio não resolveria a questão, pois os segmentos podem permanecer operando sob os mesmos endereços de rede de antes. Apesar de ser este um problema de difícil solução, não sendo resolvido pelo Bouncer, burlar o controle de licenças desta forma pode não trazer uma relação custo/benefício satisfatória para o eventual contraventor, devido às sérias limitações impostas ao uso da rede pela seu voluntário particionamento. No entanto, uma vez re-unificada a rede, o Bouncer irá consolidar novamente o grupo inteiro, voltando a gerenciar de forma única a quantidade real de licenças disponíveis. No próximo capítulo discutiremos com detalhes como isto pode ser feito.

3.3.2.3 O Protocolo Bouncer Tolerante a Faltas

Uma vez detectadas as fragilidades do protocolo e obtidas as respectivas soluções para estas, vamos então acoplá-las ao protocolo Bouncer, de modo a torná-lo tolerante a tais problemas. A figura 3.3 mostra a nova versão do protocolo Bouncer, contendo tais modificações.

O servidor B entra em execução e junta-se ao grupo Bouncer. Caso o grupo não exista, B irá criá-lo. Já integrado ao grupo Bouncer, B irá solicitar a tabela de licenças ao líder do grupo.

```
1 BOUNCER =
2   [ NOGROUP → CREATE_GROUP_BOUNCER ]
3   JOIN_TO_GROUP;
4   GETTABLE( Bouncer( LEADER( Bouncer (i:1..n) ) ) );
5   *[ SEARCH_LOCAL_LICENSES( TabLicenses) ? i →
6       [DEAD( Client(i, j) ) → Bouncer(i: 1..n) ! Release_j]
7   ]
8   Client(i, j) ? License_j →
9   [ NumLicenses_j = MaxLicenses_j → AUDIT( j )]
10  [ NumLicenses_j = MaxLicenses_j → Client(i, j) ! NoLicense();
11  [] NumLicenses_j < MaxLicenses_j → Bouncer (i:1..n) ! License_j ;
12  ]
13  ||
14  Bouncer (i:1..n) ? License_j →
15  [ NumLicenses_j < MaxLicenses_j →
16      RECORD_LICENSE(j, TabLicenses);
17      [ MyClient(i) →
18          Client(i, j) ! LicenseOk();
19      ]
20  [] NumLicenses_j = MaxLicenses_j →
21      [ MyClient(i) →
22          Client(i, j) ! NoLicense();
23      ]
24  ]
25  ||
26  Client(i,j) ? Release_j → Bouncer (i:1..n) ! Release_j;
27  ||
28  Bouncer (i:1..n) ? Release_j → Cliente(i, j) ! RELEASE_LICENSE(j, TabLicenses);
29  ||
30  [ DEAD( Bouncer (i: 1..n) ) ? k → RELEASE_ALL( TabLicenses, K);
31              RESET_GROUP( k );
32  ]
33  ||
34  Client(i,j) ? Monitore_j → Client(i,j) ! MONITOR(j);
35  ||
36  Bouncer(i) ? Status → [ I_AM_LEADER → Bouncer(i) ! TabLicenses ]
```

Figura 3.3 - O protocolo Bouncer tolerante a faltas.

Uma vez obtida a tabela, B irá procurar pela existência de licenças oriundas de sua própria máquina. Como podemos ver nas linhas de 5 a 7 da figura 3.3, a cada ocorrência encontrada, será verificado se a respectiva aplicação cliente ainda

está ativa. Caso não esteja, um pedido de liberação da licença será enviado ao grupo **(S1)**.

Quando um pedido de licença é recebido por B, este verifica em sua tabela a disponibilidade de licenças para tal aplicação. Caso não haja mais licenças disponíveis, um processo de auditoria será iniciado no grupo. Sendo confirmada a utilização de todas as licenças disponíveis, a aplicaçao terá seu pedido de licença negado (linha 10 da figura 3.3). Havendo alguma licença disponível, o pedido será repassado para o grupo Bouncer **(S2)**.

Ao receber um pedido de licença do grupo, B fará uma busca por licenças disponíveis para aquela aplicação em sua tabela. Havendo licença disponível, esta será cadastrada na tabela, dando-lhe o *status* de "em uso". O B que enviou o pedido ao grupo irá, então, informar ao seu cliente que a licença foi concedida e que a aplicação pode prosseguir com sua execução. Se não houver mais nenhuma licença disponível, este mesmo B negará o pedido feito por seu cliente (linhas de 14 a 24 da figura 3.3).

Quando uma liberação de licença é recebida de um cliente, B a repassa para o grupo. Só as liberações recebidas do grupo serão processadas pelo B que enviou a mensagem. Isto garante que todo o grupo também recebeu a mensagem e, portanto, as tabelas serão mantidas consistentes. Ao receber uma liberação de licença do grupo, aí sim, B remove a entrada referente a esta licença de sua tabela (linhas 26 e 28 da figura 3.3).

Caso seja detectado que um membro se desligou (voluntariamente ou devido a alguma falha) do grupo, este é refeito, sendo excluído o membro ausente e todas as licenças de sua respectiva máquina serão liberadas (linhas 30 a 32 da figura 3.3). Esta liberação é feita localmente, de forma independente por cada B **(S1)**.

Caso receba um pedido de monitoração, B devolverá ao cliente uma estrutura de dados preenchida com o conteúdo de sua tabela de licenças (linha 34 da figura 3.3).

B, caso seja líder de grupo, está apto a enviar sua tabela de licenças a servidores recém-engajados ao grupo, como forma de fornecer o estado atual de utilização das licenças controladas pelo Bouncer. Isto é feito pelo último bloco de comandos do código CSP apresentado na figura 3.3 (linha 36).

Finalmente, todas as mensagens trocadas tanto pelo grupo de processos Bouncer quanto pelos processos B e suas respectivas aplicações clientes serão devidamente criptografadas, o que dificulta a introdução de um falso servidor B **(S3)**.

3.3.3 O Processo de Auditoria de Licenças Órfãs

Uma licença órfã é aquela cuja respectiva aplicação teve sua execução encerrada de maneira anormal, não tendo liberado a licença que até então estava consumindo. Desta forma, apesar desta aplicação não mais estar em execução, sua antiga licença ainda está com o *status* de "em uso" para o grupo Bouncer.

O processo de auditoria consiste em varrer as tabelas de licenças mantidas pelos processo B em busca de licenças órfãs, liberando-as para poderem ser usadas por outros usuários.

Cada Bouncer mantém uma tabela de licenças em sua respectiva máquina, baseado nas mensagens recebidas do grupo de processos do qual faz parte. As propriedades de distribuição das mensagens especificadas anteriormente irão garantir que todas as tabelas de licenças dos processos Bouncer estejam consistentes entre si. Como foi visto na descrição do protocolo, cada Bouncer, ao receber um pedido de auditoria, se encarrega de checar em sua tabela local as entradas referentes às licenças utilizadas em sua respectiva máquina, enviando eventuais pedidos de liberação para o grupo. A liberação de uma licença só é realmente processada pelo autor do pedido quando a mensagem chega do grupo. Isto é feito para garantir a consistência das informações dentro do grupo.

O Bouncer que toma a iniciativa da auditoria é aquele que está prestes a negar um pedido de licença a um cliente, por haver constatado que todas as licenças para a aplicação em questão estão sendo utilizadas. Para ter certeza de que esta conclusão não está se baseando em informações que não refletem a realidade, a auditoria é feita primeiramente no âmbito local. Ao finalizar sua auditoria local, o Bouncer irá enviar uma mensagem para o grupo indicando **Auditoria Feita (AF)**. Cada membro do grupo, então, ao receber esta mensagem, inicia sua auditoria, enviando a mensagem AF ao grupo após sua conclusão. Em suma, uma licença só é negada quando o resultado de uma auditoria distribuída não libera nenhuma licença, e nenhuma licença é liberada espontaneamente durante este processo de auditoria.

Uma questão logo surge. Quanto tempo o Bouncer que iniciou a auditoria deverá esperar por liberações de licenças de outros membros do grupo para responder ao seu cliente? Uma alternativa é estabelecer um tempo de espera, após o qual o Bouncer negará ou aprovará a execução da aplicação cliente. Outra maneira, mais transparente, é esperar até que se tenha a certeza de que todos os membros do grupo enviaram sua mensagem AF. Isto seria possível com o auxílio dos recursos de gerência do grupo Bouncer comentados anteriormente. Só que, para que esta detecção seja possível, é necessário que o mecanismo de controle do grupo verifique a integridade dos seus membros a cada mensagem entregue, o que poderia tornar a comunicação consideravelmente lenta.

A situação pode se agravar quando um usuário tenta insistentemente utilizar uma aplicação cujas licenças estão realmente esgotadas. Isto poderia causar

sucessivos e desnecessários processos de auditoria, o que poderia prejudicar o desempenho do sistema e da rede.

Como pode ser notado, o processo de auditoria distribuída, como descrito acima, pode afetar consideravelmente o desempenho do serviço, apesar de fornecer os resultados mais fiéis. Entretanto, é possível usar um processo de auditoria alternativo bem mais eficiente em termos de desempenho, com a desvantagem de, em alguns casos, vir a gerar algumas recusas indevidas de licença. A seguir, descreveremos esta alternativa.

a) As auditorias são sempre realizadas localmente (cada processo B verifica a validade de suas aplicações em sua respectiva tabela) e independentes de auditorias executadas em outros processos do grupo.

b) Duas auditorias não podem acontecer dentro de um intervalo de tempo menor do que um certo valor T.

c) Os seguintes eventos podem levar à execução de uma auditoria local (dependendo exclusivamente do cumprimento do item b, acima):

c.1) concessão da última licença disponível;

c.2) pedido de licença recebido do grupo, negado porque todas as licenças estão alocadas.

Esse protocolo negaria uma licença indevidamente apenas nos seguintes casos:

1. após a realização de uma auditoria, uma ou mais aplicações falharam sem liberar suas licenças. Esta auditoria foi provocada por conta da alocação da última licença disponível para esta aplicação. Nesse meio tempo, nenhuma aplicação liberou nenhuma licença nem teve um pedido de licença negado (o que seria motivo para a realização de uma nova auditoria, caso não violasse o ítem b). Neste cenário, uma aplicação solicita uma licença.
2. O mesmo cenário de 1; e o pedido de licença negado em 1 ocorreu menos de T unidades de tempo depois da auditoria feita por conta da última licença ter sido alocada; e uma aplicação solicita uma licença.

Note que se o ítem b) for relaxado, haverá uma única situação (a situação 1, descrita acima) em que uma licença será negada indevidamente. Além disso, um novo pedido de licença executado logo em seguida tem grandes chances de ser aceito, e uma chance muito pequena de gerar uma negação indevida de licença.

Nós acreditamos que esta é a melhor solução, uma vez que os processos de auditoria não serão propagados para o grupo inteiro em um único momento, provocando uma grande operação coletiva do grupo, mas serão realizadas assíncrona e isoladamente por cada processo B. Isto, além de evitar tráfego intenso

na rede, concentrado durante os processos de auditoria, torna a resposta ao cliente mais ágil e sem a necessidade de se estabelecer um *time-out* para a recepção de respostas dos membros do grupo.

Capítulo 4

4 Infra-estrutura de Comunicação do Bouncer

Neste capítulo, faremos uma série de considerações sobre a forma de implementação dos mecanismos de comunicação entre processos (*Inter-Process Communication* - IPC) no Bouncer. Como visto anteriormente, no serviço Bouncer, vários processos precisam se comunicar. Processos que representam as aplicações protegidas pelo Bouncer têm que se comunicar com o *daemon* servidor Bouncer, que por sua vez deve se comunicar com outros *daemons* servidores Bouncer espalhados pelas diversas máquinas da rede local (vide figura 3.1 da página 28).

4.1 Comunicação entre a Aplicação e o Servidor Bouncer

Quando uma aplicação solicita uma licença de execução ao servidor Bouncer, este irá verificar a disponibilidade de tal licença e enviará uma resposta à aplicação com o resultado deste levantamento. Nesta fase, está havendo uma comunicação entre dois processos do tipo pedido-resposta, caracterizando o modelo cliente-servidor.

4.1.1 O Modelo Cliente-Servidor

O modelo C-S é uma alternativa de comunicação entre processos bastante simples e intuitiva. Duas entidades desejam se comunicar, de forma que uma delas tem um problema a ser resolvido e necessita da prestação de um determinado serviço pela outra entidade. Chamemos a primeira entidade de cliente e a segunda de servidor. O cliente envia uma mensagem para o servidor contendo a solicitação do serviço que será provido pelo servidor. Uma vez processado este pedido, uma outra mensagem é encaminhada pelo servidor com destino ao cliente, contendo o resultado do processamento do seu pedido. Este resultado pode se apresentar na forma de dados ou simplesmente como o *status* da operação realizada pelo servidor.

O modelo C-S é uma alternativa de comunicação simples, que veio fazer frente aos protocolos baseados no modelo em camadas [Tanembaum 89]. As constantes inserções e remoções de informações de controle (cabeçalhos) aos dados a serem trocados entre processos, características do modelo em camadas, podem afetar consideravelmente o desempenho do ambiente onde esta comunicação está se dando. Em redes geograficamente distribuídas (WANs), isto não trás grande sobrecarga, pois, devido às baixas taxas de transmissão dos canais de comunicação, que cobrem grandes distâncias, as CPUs comunicantes conseguem processar estes cabeçalhos e ainda manter o canal trabalhando em sua capacidade total. Portanto, o modelo em camadas pode ser utilizado sem perda de desempenho neste tipo de ambiente. Entretanto, para ambientes de redes locais (LANs), como os canais de transmissão em LANs são mais rápidos, a sobrecarga com o processamento dos cabeçalhos é substancial, havendo um sub-aproveitamento da vazão que a LAN pode oferecer [Tanembaum 92b].

Ainda visando ganhar eficiência, geralmente o modelo C-S procura evitar a considerável sobrecarga com o estabelecimento de conexões, optando por uma comunicação não orientada à conexão. Desta forma, a eficiência é uma consequência direta da simplicidade.

Na implementação de um serviço de comunicação C-S, existem quatro importantes aspectos a serem considerados: endereçamento, bloqueio, buferização e confiabilidade. Passaremos agora a discutir cada um destes ítens, apresentando as alternativas oferecidas ao projetista de tal serviço.

4.1.1.1 Endereçamento

Para que um cliente possa enviar uma mensagem para um servidor, ele deve saber o endereço do processo destino, o mesmo acontecendo com o servidor, ao enviar uma mensagem no sentido contrário. É razoável que sejam utilizados, neste caso, os identificadores destes processos para endereçar mensagens trocadas entre eles (por exemplo, o processo cliente 234 troca mensagens com o servidor cujo identificador é 36). Entretanto, estes processos podem estar situados tanto na mesma máquina quanto em máquinas distintas de uma rede. Na segunda hipótese, vários processos espalhados pela rede podem possuir identificadores iguais, gerando ambigüidade no momento da entrega de uma mensagem. Digamos que um cliente localizado em uma máquina chamada *host1* deseje enviar uma mensagem para um processo servidor cujo identificador é 102 e está localizado em outra máquina, chamada *host2.* Existem, porém, dois processos com este mesmo identificador, um, o servidor desejado, localizado na máquina *host2* e outro, um processo qualquer, na máquina *host3.* Esta ambigüidade é resolvida quando o esquema de endereçamento *processo.máquina* é adotado. No nosso exemplo, o servidor seria corretamente referenciado através do endereço *102.host2.* O problema com esta abordagem é que os clientes têm que necessariamente saber em que máquina da rede um determinado servidor se encontra. Caso esta máquina seja desativada e o servidor transferido para um outro *host*, os clientes deverão ser

atualizados, sob pena de não conseguirem localizar o servidor. Isto não é adequado, já que a transparência é uma das principais metas a serem atingidas no projeto de serviços deste tipo.

Outra alternativa de endereçamento é garantir uma identificação única para os processos servidores no âmbito de uma rede, onde um grande espaço de endereçamento estaria disponível. Em redes que suportem comunicação por difusão (*broadcast*), um serviço seria então localizado pelos clientes através da difusão de uma mensagem de busca contendo o identificador único do processo servidor desejado. O servidor, então, responderia a esta mensagem com o endereço da máquina onde está localizado. A partir daí, cliente e servidor trocariam suas mensagens normalmente, sem a necessidade de novos *broadcasts* (o cliente guarda o endereço do servidor consigo). O inconveniente desta forma de endereçamento é o constante envio de mensagens por *broadcast,* o que gera sobrecarga para as aplicações que monitoram a rede.

Uma terceira abordagem é identificar serviços através de nomes de alto nível (ex.: cadeias de caracteres ASCII). Um processo localizado em uma máquina bem conhecida da rede fica responsável por mapear nomes de serviços em endereços de máquinas nas quais estes podem ser encontrados. Este método chama-se Serviço de Nomes. Portanto, um cliente que deseja localizar um determinado servidor através do seu nome, envia uma consulta ao servidor de nomes, que devolve a localização do respectivo servidor. A partir deste ponto, cliente e servidor irão se comunicar de forma direta. O serviço de nomes possui um inconveniente, comum a todos os serviços centralizados: ter um único ponto de falha. Caso o servidor de nomes venha a falhar, todos os serviços que dependem dele para serem localizados estarão indisponíveis para seus respectivos clientes. Este problema pode ser contornado através da replicação do serviço de nomes.

4.1.1.2 Primitivas Bloqueantes x Primitivas não Bloqueantes

Para que dois processos possam trocar informações, existem primitivas (funções específicas) que realizam esta tarefa. Estas primitivas de troca de mensagens são classificadas em: bloqueantes e não bloqueantes.

As primitivas bloqueantes, também conhecidas como *primitivas síncronas*, funcionam da seguinte forma: quando um processo chama uma função de envio de mensagem (*send*), este é bloqueado até que a mensagem armazenada no *buffer* passado como parâmetro para a função seja completamente enviada. De maneira similar, a chamada a uma função de recepção de mensagens (*receive*) não retornará o fluxo de controle à próxima instrução até que tenha concluído a recepção da mensagem, armazenando-a no *buffer* passado como parâmetro da função. A vantagem da utilização de primitivas síncronas é que, uma vez concluídas as operações de envio ou recepção de mensagens, os conteúdos dos *buffers* por elas ocupados podem ser re-utilizados. Em contrapartida, enquanto um processo está bloqueado enviando ou recebendo dados de maneira síncrona, seu tempo de

resposta aumenta, uma vez que este está impedido de processar outras atividades enquanto transmite ou recebe mensagens.

Já as primitivas não bloqueantes, ou *primitivas assíncronas,* liberam o processo para continuar sua execução em paralelo com a transmissão ou recepção das mensagens, o que pode gerar um considerável ganho de desempenho na execução dos processos. A desvantagem deste tipo de primitivas é que os *buffers* de transmissão ou recepção não podem ser modificados pela aplicação até que se tenha certeza de que sua manipulação foi concluída. Isto deve ser feito para evitar que as informações do *buffer* sejam danificadas/sobrepostas. Também ocorre que, no envio de uma mensagem, como a aplicação não espera pela conclusão desta operação, esta não tem idéia de quando a transmissão será concluída. Uma solução para isto seria copiar os conteúdos dos *buffers* para o núcleo do sistema, liberando-os para serem re-utilizados. As primitivas, então, manipulariam diretamente as cópias dos *buffers.* Porém, esta cópia extra pode reduzir sobremaneira o desempenho do sistema.

4.1.1.3 Primitivas Buferizadas x Primitivas não Buferizadas

Assim como é possível optar entre primitivas de comunicação bloqueantes e não bloqueantes, existem também disponíveis as opções de primitivas *com* buferização e sem buferização.

Quando um processo servidor executa uma primitiva de recepção de mensagem, como *receive(addr, &msg)*, ele está informando ao núcleo do sistema operacional de seu *host* que está aguardando uma mensagem enviada para o endereço *addr* e que esta mensagem deverá ser copiada pelo núcleo para o *buffer msg.* Portando, quando um determinado cliente executa uma primitiva de envio de mensagem *send(addr, msg)*, onde *addr* está referenciando o servidor em questão, esta mensagem deverá ser recebida com sucesso pelo processo servidor. Isto irá funcionar sem problemas caso o servidor ative a primitiva *receive* antes que o cliente envie sua mensagem, através da primitiva *send.* Caso isto não aconteça, ou seja, se a mensagem for enviada antes que o servidor esteja pronto para recebê-la, o núcleo do sistema operacional do servidor não saberá a quem entregar a mensagem recebida para o endereço especificado pelo cliente, nem para onde copiar esta mensagem. Isto acarretará, portanto, no descarte da mensagem. Note que esta indisponibilidade do servidor para receber mensagens pode ser momentânea, estando ele processando um pedido recebido de um outro cliente a poucos instantes atrás. Neste intervalo de tempo, um cliente pode enviar várias vezes uma mensagem, sem obter qualquer indicação de que esta foi recebida com sucesso, o que pode levá-lo a concluir que o servidor falhou ou que o serviço não está disponível naquele endereço. Este é o comportamento apresentado por primitivas que não utilizam buferização.

Uma estratégia para minimizar este problema é instruir o sistema operacional do servidor a acumular as mensagens destinadas a um certo processo em uma

área chamada **caixa postal**. Esta caixa postal deve ser criada pelo processo receptor de mensagens, indicando o endereço cujas mensagens a ele destinadas devem ser acumuladas. Desta forma, quando um processo não estiver apto a receber uma mensagem naquele endereço, o núcleo do sistema irá enfileirá-la. Na próxima execução da primitiva *receive(addr, &msg)*, a primeira mensagem enfileirada será entregue ao processo receptor. Caso *receive* seja ativada e a caixa postal estiver vazia, o processo receptor ficará bloqueado aguardando novas mensagens, se a primitiva utilizada causar este comportamento (primitivas bloqueantes). Primitivas que se comportam assim utilizam buferização.

Existe, no entanto, um limite para o crescimento das caixas postais. Quando este limite for atingido, as mensagens subseqüentes a este evento serão descartadas, ou seja, as primitivas com buferização agirão de forma idêntica às primitivas sem buferização. Uma solução para isto é não permitir que processos enviem mensagens para um destino que não tenha capacidade de armazená-las. Quando houver novamente a possibilidade de recepção de mensagens, isto será indicado para os processos que desejem enviá-las. Para tal, faz-se necessário implementar um mecanismo de controle de fluxo de mensagens [Macedo 95].

4.1.1.4 Primitivas Confiáveis x Primitivas não Confiáveis

Até agora, foi assumido que as mensagens trocadas entre processos cliente e servidor sempre chegam com sucesso ao seu destino. Porém, no mundo real, estas mensagens estão sujeitas a se extraviar. Nesta situação, um cliente, após o envio de uma mensagem ao servidor, não tem garantias de que sua mensagem chegou ao destino. Da mesma forma, quando um servidor envia resultados ao cliente, não está certo de que estes resultados foram corretamente recebidos. Existem três diferentes abordagens para este problema.

Na primeira abordagem, é assumido que a primitiva de envio de mensagens *send* não é confiável. O sistema não dá nenhuma garantia sobre a correta entrega das mensagens. Desta forma, é deixado sob a responsabilidade do usuário implementar esta confiabilidade. Tomemos como analogia o sistema de correio convencional. A companhia de correios fará o melhor possível para entregar uma correspondência, porém, não dá nenhuma garantia de que esta realmente chegará ao seu destino. Esta garantia pode ser oferecida através de um serviço extra.

A segunda abordagem é fazer com que o receptor envie um reconhecimento ao transmissor de que a mensagem foi recebida com sucesso. É estipulado um período de tempo para que isto ocorra. Caso este tempo expire, o transmissor assume que a mensagem foi extraviada, retransmitindo-a. Este mecanismo de reconhecimento é transparente para os processos, pois está embutido nas primitivas de comunicação.

A terceira abordagem é tirar proveito da característica de comunicação do modelo C-S que consiste em uma chamada a um servidor com a espera de algum

retorno deste. Neste caso, a própria resposta do servidor ao pedido do cliente funcionaria como um reconhecimento. Desta forma o cliente recebe uma confirmação da entrega de sua mensagem, porém isto não ocorre com o servidor, que não recebe nenhuma garantia de que os resultados foram recebidos pelo cliente. Em certas situações isto se faz necessário. Quando um pedido de um cliente gera um processamento pesado por parte do servidor para atender esta solicitação, é interessante que o servidor, ao enviar os resultados para o cliente, não descarte estes resultados até ter certeza de que eles foram recebidos com sucesso. Isto evitaria a necessidade de re-processar o pedido do cliente.

4.1.2 Remote Procedure Call

O modelo C-S oferece uma maneira bastante conveniente para a implementação de comunicação entre processos em sistemas distribuídos, sendo amplamente utilizado. Porém, este paradigma deixa a desejar no que diz respeito à transparência para o programador, pois a forma de implementar a troca de mensagens entre processos é bastante arraigada em operações de E/S, devido ao uso explícito de primitivas como *send* e *receive*, por exemplo. Isto torna-se um inconveniente por exigir um esforço maior de programação, já que a complexidade inerente à computação distribuída é consideravelmente maior que a associada à computação centralizada.

Birrell e Nelson propuseram uma maneira totalmente nova de atacar este problema [Birrell 84]. A idéia, bastante simples e intuitiva, consiste em ativar procedimentos localizados em outras máquinas de uma rede de forma idêntica à ativação de procedimentos locais. Utilizando este princípio, um processo executando em uma máquina A poderá ativar um procedimento implementado em uma máquina B. Dados são trocados entre os processos na forma de parâmetros e de resultados da execução do procedimento, de forma totalmente transparente ao programador. Nenhuma mensagem é enviada ou recebida através da utilização de primitivas de E/S. Nos próximos tópicos apresentaremos importantes características do paradigma de ativação remota de procedimentos (*Remote Procedure Call* ou RPC).

4.1.2.1 Princípios Básicos de Operação do RPC

A idéia por trás do RPC é tornar uma chamada a um procedimento localizado em outra máquina o mais semelhante possível a uma chamada local.

Quando um procedimento local é ativado, seus parâmetros são armazenados em uma estrutura de dados mantida pelo sistema operacional denominada pilha de execução. O fluxo de controle é, então, desviado para o código executável do procedimento, que desempilha os parâmetros, executa sua funcionalidade e armazena seus resultados onde estes possam ser recuperados (registradores, por exemplo). A chamada ao procedimento é finalmente concluída com o

desempilhamento do endereço de retorno, para onde o fluxo de controle é novamente desviado e os resultados são recuperados.

Para que seja entendido o funcionamento da chamada a um procedimento remoto, é necessário que antes sejam apresentados novos elementos que irão participar deste processo. Estes elementos são o ***stub*** **cliente** e o ***stub*** **servidor.** Os *stubs* compõem uma camada de software que é responsável por tornar transparente a comunicação entre os processos. Eles é que efetivamente fazem uso das primitivas de comunicação para a troca de mensagens entre cliente e servidor, além de manipular os parâmetros e resultados dos procedimentos.

O *stub* cliente encontra-se ligado ao código do cliente. Nele está contida apenas a interface dos procedimentos remotos. Portanto, quando um procedimento é ativado, para a aplicação cliente, é como se este estivesse implementado dentro do *stub.* Uma vez recebidos os parâmetros, estes são encapsulados em mensagens e enviados ao processo servidor através da primitiva *send.* Imediatamente após o envio dos dados para o servidor, o *stub* cliente chama a primitiva *receive,* o que irá causar o bloqueio do processo até que uma mensagem contendo o retorno do servidor seja recebida.

O *stub* servidor, por sua vez, está ligado com uma biblioteca composta pelos procedimentos a serem executados, funcionando como um direcionador para o procedimento executado a partir do cliente. O *stub* servidor está constantemente bloqueado por uma chamada à primitiva *receive* até que uma mensagem seja recebida de algum cliente, contendo a ativação de um procedimento. Esta mensagem é, então, desencapsulada, de onde são extraídos o identificador do procedimento a ser ativado e seus parâmetros. Um *switch* é executado a partir deste identificador, redirecionando o fluxo de execução para o respectivo procedimento. Neste momento, o stub servidor está assumindo o papel da aplicação cliente. Quando finalmente os resultados forem disponibilizados, estes serão novamente encapsulados em uma mensagem a ser enviada ao cliente através da primitiva *send.*

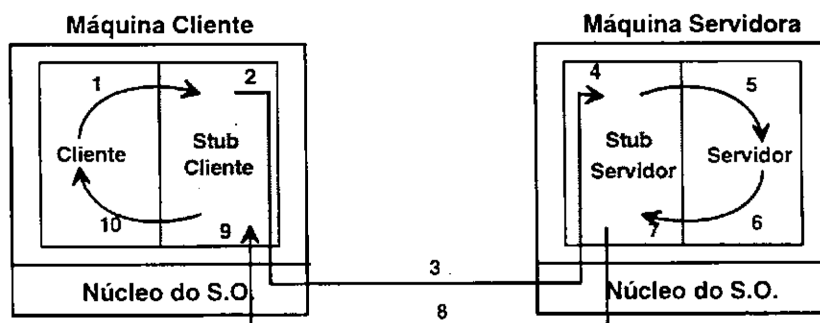


Figura 4.1 - Seqüência de ativação de procedimentos remotos

A figura 4.1 sumariza toda a seqüência de ativação de um procedimento remoto. Seguindo os passos indicados, temos:

1. O cliente chama o procedimento localizado no *stub* cliente.
2. O *stub* cliente constrói uma mensagem contendo os parâmetros e, através da primitiva *send*, a entrega ao núcleo do sistema operacional da máquina cliente. Imediatamente ativa *receive* e aguarda resposta do servidor.
3. O núcleo do sistema operacional cliente, através da rede, entrega a mensagem ao núcleo do sistema operacional da máquina do servidor.
4. O núcleo do sistema operacional servidor entrega a mensagem ao *stub* servidor, que até então estava bloqueado após ativar a primitiva receive.
5. O *stub* servidor extrai os parâmetros da mensagem e ativa o procedimento desejado.
6. O servidor executa o procedimento, entregando os resultados ao *stub* servidor.
7. O stub servidor encapsula os resultados em uma mensagem e a entrega ao núcleo do sistema operacional local através da primitiva *send*.
8. O núcleo do sistema operacional servidor envia os resultados ao núcleo do sistema operacional cliente.
9. O núcleo do sistema operacional cliente entrega a mensagem ao *stub* cliente.
10. O stub extrai os resultados da mensagem recebida e os entrega à aplicação cliente.

Desta forma, nem o cliente nem o servidor tomam conhecimento dos passos intermediários, devidamente ocultados pelos *stubs*.

4.1.2.2 Passagem de Parâmetros

Como visto, a função do *stub* cliente é receber os parâmetros do procedimento chamado, empacotá-los em uma mensagem e enviá-la para o *stub* servidor. Também é sua função receber a mensagem contendo os resultados da ativação do procedimento e entregá-los à aplicação cliente. A funcionalidade simétrica é realizada pelo *stub* servidor. À primeira vista, estas parecem ser operações bastante triviais, porém envolvem certos detalhes que devem ser considerados.

Em primeiro lugar, um servidor pode estar ligado a vários procedimentos remotos. Portanto, é necessário indicar na mensagem construída pelo *stub* cliente qual o procedimento que deverá ser executado.

Em segundo lugar, as arquiteturas das máquinas onde estão executando o cliente (máquina cliente) e o servidor (máquina servidora) podem não ser idênticas. Assim sendo, o mecanismo de RPC deve prover alguma forma de tornar compatível a maneira como os dados são representados nestas máquinas. Tomemos como exemplo uma máquina com arquitetura Intel [Intel 96] e uma máquina com arquitetura RISC SPARC [Sun 96]. Neste caso, a maneira como os bytes de um número inteiro são dispostos em uma arquitetura é o inverso da outra. Para solucionar este problema, foi adotado um esquema de conversão dos dados para uma forma única de representação enquanto estes estiverem circulando através da rede. Esta forma é conhecida como **Forma Canônica** ou ***Network Byte Order*** (NBO) [SCO 95]. As representações adotadas por cada arquitetura em particular são conhecidas como ***Host Byte Order*** (HBO). Portanto, quando uma mensagem é construída pelo *stub* cliente, este realiza antes uma conversão da representação HBO de sua máquina para a NBO, que é universal. Ao chegar no *stub* servidor, esta mesma mensagem será convertida de NBO para a representação HBO local do servidor.

É de suma importância observar que o processo de compatibilização de representações não se trata simplesmente de uma inversão na ordem dos bytes, pois quando se trata de caracteres, não há a necessidade desta inversão. Portando, é necessário também que, ao encapsular os parâmetros dos procedimentos, seja indicado o tipo do dado que está sendo encapsulado, para que este tenha o tratamento específico pelo processo de conversão de representação. O RPC utiliza protocolos específicos para realizar esta tarefa. Através deles é possível a construção de rotinas para linearizar e deslinearizar os dados transmitidos e recebidos, respectivamente, através da rede. Como exemplos deste tipo de protocolo podemos citar o *External Data Representation* (XDR) [Sun 88] e o Linear [Drummond 96]. O processo de linearização/deslinearização consiste em converter todos os tipos de dados para cadeias de caracteres, já que estes possuem uma ordenação de bytes compatível em todas as arquiteturas.

Uma última consideração sobre a passagem de parâmetros e, talvez, a mais importante, diz respeito ao tratamento de variáveis do tipo apontador. Como sabemos, um apontador para um determinado dado consiste em um inteiro longo que denota o endereço de memória onde este dado está armazenado. É notório que este apontador só fará sentido no seu espaço de endereçamento. Então, como serão tratados parâmetros deste tipo para a ativação de procedimentos remotos? Um apontador para uma cadeia de caracteres na linguagem C, por exemplo, ao ser passado como parâmetro para um procedimento que executará no espaço de endereçamento do servidor, poderá estar referenciando o núcleo do sistema operacional, o código executável do servidor ou a pilha de execução de uma aplicação qualquer. Portanto, um desastre pode ocorrer caso este endereço seja utilizado no servidor. A solução para isto é, do lado cliente, fazer uma copia da área referenciada pelo apontador para um vetor de caracteres e transmiti-lo normalmente. Uma vez no servidor, após devidamente deslinearizado, este vetor já estaria alocado no espaço de endereçamento do servidor. Um apontador para esta

área seria entregue, então, ao procedimento. As modificações feitas seriam, portanto, retornadas no momento de linearizar este vetor e enviá-lo ao cliente. Ao ser deslinearizado, o resultado seria re-mapeado para a área originalmente referenciada pelo apontador, devidamente atualizada. É importante observar que o processo de linearização/deslinearização será recursivo, caso a área referenciada por um apontador contenha também referências a outras áreas de memória. O leitor atento irá notar que esta solução possui uma limitação, que é a necessidade de um prévio estabelecimento do tamanho da área referenciada pelo apontador. Este tamanho deverá ser fornecido como um dos parâmetros para o procedimento, para que seja alocado um vetor de caracteres suficientemente grande para armazenar o dado.

4.1.2.3 *Binding* Dinâmico

Como já discutido na seção que trata do modelo C-S, uma questão que também se aplica ao RPC é a de como um servidor é localizado na rede. Uma das possíveis formas seria ter esta localização estaticamente indicada no código do cliente, o que, como já visto, é uma alternativa inviável dada a sua inflexibilidade. Os servidores podem mudar sua localização mediante situações de falha em processos ou máquinas, ou simplesmente pela necessidade de manutenção em equipamentos.

Os serviços baseados em RPC utilizam um mecanismo similar ao serviço de nomes, também comentado anteriormente, denominado ***dynamic binding***, que passaremos a descrever neste tópico. Antes, porém, devemos comentar algo a respeito de como estes servidores são endereçados.

Existe uma etapa que deve ser cumprida para que se possa disponibilizar serviços implementados com RPC, que é a sua especificação através de uma linguagem apropriada para este fim. Esta especificação é composta por informações como: nome do serviço, sua versão, um identificador único e a descrição de sua interface (uma lista de procedimentos oferecidos pelo servidor, com a descrição dos parâmetros e tipos de retorno para cada um deles, inclusive definições de tipos de dados feitas pelo programador). Normalmente esta descrição é feita em um arquivo texto, que é processado por um compilador específico para então gerar arquivos contendo definições de estrutura de dados, *stubs* cliente e servidor e filtros de linearização/deslinearização de parâmetros.

Quando um processo servidor (biblioteca de procedimentos + *stub* servidor), entra em execução, este exporta sua interface, o que significa registrar-se junto a um programa chamado ***binder***. Este registro consiste em informar ao *binder* qual o identificador do serviço prestado e sua localização na rede. O *binder* possui uma localização bem conhecida, de modo a ser encontrado sem problemas pelos servidores. Uma vez registrado, um servidor está apto a ser localizado por qualquer cliente que interpele o *binder* a seu respeito. Da mesma forma, quando um servidor

está apresentando problemas e tem que finalizar sua execução, pode cancelar seu registro junto ao *binder*.

Ao utilizar um serviço pela primeira vez, ou seja, ao chamar pela primeira vez um dos procedimentos disponibilizados por um servidor, um cliente ainda não realizou o *bind* com este servidor (sua localização ainda é ignorada pelo cliente). O *binder* é, então, consultado através de uma chave de pesquisa (por exemplo, nome_do_serviço + versão + identificador_único), obtendo um *handle* para o servidor (no caso de redes TCP/IP, o *handle* é formado pelo endereço IP + porta) que é retornado para o cliente. A partir de então, este *handle* é armazenado pelo cliente para que este não tenha que realizar novamente um *bind* para um servidor já localizado.

Uma desvantagem do esquema de *dynamic binding* é o caso de clientes com vida curta. Isto faz com que muitas consultas ao *binder* sejam feitas, o que irá gerar um tráfego, muitas vezes, significativo na rede.

É interessante que em grandes redes, onde muitos serviços são disponibilizados e um grande número de clientes procura por estes serviços, vários *binders* sejam disponibilizados, seja para evitar gargalos na rede (acesso intenso ao nó onde o binder está executando), seja para implementar tolerância a faltas no *binder* (isto é importante, pois nenhum serviço dependente do *binder* será localizado caso este falhe).

4.1.2.4 Semântica do RPC na Presença de Falhas

A meta do RPC é tornar a comunicação entre processos baseada no modelo C-S transparente. Exceto por alguns pontos, como a manipulação de variáveis globais e tratamento de parâmetros do tipo apontador (passagem por referência), estes objetivos são alcançados, desde que clientes e servidores estejam em perfeito funcionamento. Quando as situações de falha começam a acontecer, torna-se difícil manter esta transparência. A seguir discutiremos os principais problemas que podem ocorrer com serviços baseados em RPC, discutindo cada um deles.

_i_) O cliente não consegue localizar o servidor

Várias situações podem levar a este tipo de falha. Uma delas é o fato do servidor realmente não estar disponível, pois falhou por algum motivo. Outra possível causa seria uma aplicação utilizando um *stub* cliente de uma versão antiga do serviço. O servidor ganhou novas funções em sua interface, gerando novos *stubs* e identificação de registro. O cliente antigo, então, tentará sem sucesso localizar o serviço junto ao *binder*. A não localização do serviço pode também se dar pela indisponibilidade do próprio *binder*.

ii) **Requisições de serviços perdidas**

Uma mensagem enviada a um servidor pode não chegar ao seu destino por diversas razões, desde falhas na rede até o endereçamento incorreto de um serviço. Quando o cliente envia uma mensagem, um temporizador é inicializado. Se este temporizador expirar antes que qualquer confirmação seja recebida, a mensagem será retransmitida. Caso a mensagem original tenha sido realmente perdida, nenhum efeito colateral é gerado junto ao servidor com a retransmissão da mesma. A outra situação possível será analisada a seguir.

iii) **Respostas perdidas**

Esta situação de falha é mais difícil de ser tratada que a anterior. Para o cliente, quando uma confirmação do servidor não for recebida até que seu temporizador expire, este não sabe determinar se a mensagem por ele enviada não chegou, se chegou com sucesso e a respectiva confirmação se perdeu, ou se o problema está na lentidão do servidor ou da rede. Indiferentemente, o cliente irá retransmitir a mensagem. Retransmissões indevidas, em certos tipos de operação, podem causar efeitos colaterais desastrosos.

Certas operações podem ser repetidas inúmeras vezes, sem que isto traga problemas mais sérios. Por exemplo, a transferência de um arquivo pode ser requisitada a um servidor quantas vezes for necessário para que este chegue com sucesso ao cliente. Estas operações são ditas **idempotentes**.

Em contrapartida, existem outras operações cujo número de vezes que são realizadas é relevante. Por exemplo, um cliente solicitou a um servidor a transferência de um certo valor de sua conta bancária para uma outra. O servidor deverá confirmar o sucesso da transação para o cliente. Caso esta confirmação se perca e o cliente retransmita a solicitação de transferência, o reprocessamento irá resultar em uma nova transferência de fundos. Estas operações são ditas **não idempotentes**. Para evitar os efeitos colaterais gerados pelo reprocessamento de operações deste tipo, uma solução é fazer com que o cliente numere suas requisições de serviços. Por outro lado, os servidores devem manter um registro do pedido mais recentemente processado. Desta forma, caso haja perda de confirmação e conseqüente retransmissão do pedido, o servidor terá condições de concluir que se trata de uma retransmissão e que esta não deve ser novamente processada.

iv) **Falha do servidor**

O problema de falha do servidor se assemelha ao discutido anteriormente, porém não é possível resolvê-lo com o simples seqüenciamento das mensagens. Existem duas situações distintas que envolvem falha do servidor:

1. Um pedido chega até o servidor, é processado, porém o servidor falha antes que este possa enviar o resultado de volta para o cliente.

2. O servidor recebe um pedido e, antes que possa executá-lo, falha.

Na situação 1, o núcleo do cliente não pode simplesmente retransmitir o pedido ao servidor, uma vez que a operação em questão pode não ser idempotente. Já na situação 2, a retransmissão do pedido é suficiente para resolver o problema, já que nenhum processamento foi realizado pelo servidor. A questão é: como o núcleo do cliente irá distinguir entre as duas situações?

Três diferentes semânticas de tratamento de falhas do servidor podem ser implementadas:

1. Pelo menos uma vez - O núcleo do cliente repete o pedido até que obtenha uma resposta. No mínimo uma operação foi bem sucedida, podendo ter havido outros sucessos.
2. No máximo uma vez - O núcleo do cliente tenta uma única vez acessar o servidor. Caso não obtenha sucesso, reporta o erro ao cliente. O servidor realiza, no máximo, uma operação, caso o pedido tenha chegado com sucesso. Por outro lado, se o pedido foi perdido, a operação pode não ser realizada.
3. Exatamente uma vez - Diante da inadequabilidade das duas semânticas anteriores (a semântica 1 não pode ser utilizada em operações não idempotentes; a semântica 2 não garante a execução do procedimento remoto) esta se mostra como a alternativa ideal. Porém, não é possível oferecer esta semântica de forma transparente ao programador, uma vez que não há meios de garantir que em apenas uma transmissão a mensagem será recebida pelo servidor.

v) Falha do cliente

Este problema ocorre quando um cliente faz uma chamada RPC a um servidor e falha antes que o resultado lhe seja enviado. Neste caso, temos uma computação sendo realizada no servidor que não tem para quem retornar, conhecida como órfã. Além do fato de consumir CPU na máquina servidora, esta situação torna-se inconveniente quando o cliente é re-inicializado e, inesperadamente, recebe um resultado de uma computação órfã, o que pode gerar uma certa confusão. Para evitar isto, os clientes podem proceder de três formas distintas:

1. Extermínio - O cliente mantém um log de suas chamadas RPC, de forma que, em situações de falha e re-inicialização, o log é consultado e as computações órfãs são eliminadas. A desvantagem desta abordagem é o constante acesso a disco para gravar os eventos de log.
2. Reencarnação - O cliente estabelece épocas, numeradas seqüencialmente. Após se recuperar de uma falha, um cliente envia uma mensagem por difusão informando sua época atual. Todas as computações órfãs, cujos identificadores de época estão

desatualizados, serão eliminadas pelo(s) servidor(es) que as processam.

3. Expiração - Nesta abordagem, cada chamada RPC possui uma fatia de tempo para ser concluída. Caso este tempo não seja suficiente, uma nova fatia de tempo é solicitada. A estratégia é, após uma falha do cliente, este deverá esperar um tempo, maior ou igual à fatia estipulada para as RPCs, para ser re-inicializado, de forma que será garantido que todos os órfãos terão expirado antes da re-inicialização do cliente.

4.1.3 A Utilização de RPC no Bouncer

Como dito no começo deste capítulo, a fase em que uma aplicação protegida pelo Bouncer solicita ao servidor uma licença para executar é uma situação típica de utilização do modelo C-S como paradigma de comunicação entre estes processos. Porém, pelo exposto na seção anterior, visando obter um maior grau de transparência e menor esforço de programação, o Bouncer utiliza em sua comunicação C-S o paradigma de RPC.

Os requisitos de RPC para que o Bouncer possa comunicar sua componente cliente (acoplada à aplicação do desenvolvedor) à componente servidora (*daemon* servidor B) são bastante simples, uma vez que esta comunicação não envolve o uso da rede, mas apenas a comunicação entre dois processos localizados em uma mesma máquina. Vamos analisar estes requisitos segundo os aspectos apresentados na seção anterior.

Passagem de parâmetros - O Bouncer utiliza filtros de linearização/deslinearização em suas RPCs para possibilitar a passagem de parâmetros entre os espaços de endereçamento distintos dos processos cliente e servidor. Isto se faz é necessário porque em RPC todos os parâmetros são passados por referência, ou seja, são variáveis do tipo apontador. Estes filtros, entretanto, podem ser bem mais simples, pois não existe a preocupação de compatibilidade de representação dos dados trocados entre cliente e servidor, já que se trata de uma troca de informações no âmbito de uma mesma máquina.

Binding dinâmico - Como já visto anteriormente, este ítem diz respeito a como um serviço pode ser localizado. Este procedimento envolve dois aspectos: i) como o serviço pode ser endereçado ou identificado de forma única e; ii) onde o serviço poderá ser acessado. Os serviços baseados em RPC normalmente utilizam identificadores de 32 bits. Estes identificadores devem ser únicos a nível mundial, a fim de evitar conflitos entre serviços distintos que, eventualmente, estejam utilizando um mesmo identificador. A instituição responsável por gerenciar estes identificadores é a Sun Microsystems. Portanto, quando um novo serviço baseado em RPC é disponibilizado, seja comercialmente, seja em ambiente acadêmico, desde que potencialmente possa ser utilizado globalmente, um identificador para este serviço deverá ser solicitado à Sun. Para que o *daemon* servidor Bouncer possa ser identificado de maneira única, basta que este utilize um destes

identificadores. Um detalhe importante é que o Bouncer deverá utilizar um identificador para cada kit de desenvolvimento (API + *daemon* B) que for entregue a um determinado desenvolvedor. Isto é feito para que o Bouncer seja específico por fabricante, ou seja, um Bouncer que controla as licenças dos produtos de uma *software house* A (B_A) possui um identificador de serviços distinto de outro Bouncer responsável por gerenciar aplicações de uma *software house* B (B_B). Desta forma, B_A e B_B serão essencialmente serviços diferentes, não sendo possível que as aplicações de um fabricante solicitem licenças a um Bouncer que serve licenças de um outro fabricante. Na seção 3.1 é dito que um, e apenas um servidor Bouncer estará ativo em cada máquina da rede. Note, porém, que é perfeitamente possível termos duas instâncias de servidores B, por exemplo B_A e B_B, executando em uma mesma máquina, já que B_A é diferente de B_B. Para que o Bouncer seja, então, localizado pelos clientes, não haverá a necessidade de um serviço de *binder* global, mas de um *binder* local utilizado pelo RPC, conhecido como *portmapper*. O *portmapper*, uma vez consultado sobre um determinado serviço (referenciado através de seu identificador único de 32 bits), irá retornar a porta do sistema operacional onde o respectivo servidor está aguardando ser contactado.

Tratamento de erros de RPC - O fato de que a comunicação entre cliente e servidor não faz uso da rede elimina parte dos problemas considerados anteriormente, como perda de mensagens, por exemplo. Quando mensagens são perdidas, seja no sentido cliente-servidor, seja no sentido servidor-cliente, isto gera ambigüidade na identificação do problema. No caso do Bouncer, onde a comunicação é local, não havendo, portando, a possibilidade de perda de mensagens, é fácil identificar o ponto de falha. Existem duas possibilidades de falha de comunicação durante uma chamada RPC feita pelo Bouncer: a) o *portmapper* não está ativo, logo o servidor B não poderá ser localizado; b) o *daemon* servidor B não está ativo. Na primeira hipótese, este erro será reportado pelo Bouncer para a aplicação do desenvolvedor. Já na segunda hipótese, esta falha será tratada pelo próprio protocolo Bouncer.

4.2 Comunicação entre os Servidores Bouncer

Como mostrado no capítulo anterior, o serviço Bouncer é baseado no trabalho cooperativo de vários servidores distribuídos em uma rede. Este trabalho demanda comunicação entre os processos cooperantes (*daemons* servidores B), sob a supervisão do protocolo Bouncer, também descrito no capítulo 3. Observe que, durante a especificação do protocolo Bouncer, uma série de características especiais foram pressupostas na comunicação entre os vários servidores B que compõem o serviço de licenciamento Bouncer. É necessário, portanto, que existam mecanismos para fornecer todas estas características na comunicação em grupos de processos. Iremos apresentar neste tópico um paradigma de IPC que provê estes mecanismos, denominado Comunicação em Grupo *(Group Communication)* [Birman 91] [Tanembaum 92a] [Macedo 95b].

4.2.1 Comunicação em Grupo

Na seção anterior, foram abordadas duas formas de troca de informações entre um processo que deseja um serviço (cliente) e um outro que irá prover este serviço (servidor): a utilização de primitivas de envio e recepção de mensagens (*send* e *receive*) e a chamada a procedimentos remotos (RPC). A troca de mensagens é um paradigma de IPC que implementa o envio e a recepção de mensagens entre os processos comunicantes de forma explícita, possuindo um baixo nível de abstração. O modelo de chamada a procedimentos remotos ou RPC é um dos paradigmas mais populares na computação distribuída, sobretudo em sistemas que adotam o modelo C-S. Possui um nível de abstração maior que a troca de mensagens, já que cada chamada a um procedimento remoto envolve a troca de mensagens de forma implícita: o envio da chamada ao procedimento e a recepção dos resultados (se for o caso). Em ambos os casos, tem-se apenas dois processos se comunicando por vez (comunicação 1 para 1).

Há ocasiões em que é necessário que a comunicação seja feita de 1 para muitos, ao invés da comunicação feita de 1 para 1, discutida anteriormente. Um exemplo desse tipo de comunicação é quando se tem um grupo de processos servidores trabalhando de forma cooperativa para oferecer tolerância a faltas a um determinado serviço. Nestas circunstâncias, é interessante que um cliente possa enviar uma mensagem para vários servidores. Isto irá garantir que, mesmo em caso de falha de alguns dos servidores, outros terão recebido a mensagem e poderão, portanto, processá-la.

Neste contexto, um grupo nada mais é do que uma coleção de processos trabalhando juntos para resolver um determinado problema. Estes grupos são dinâmicos, ou seja, são criados e destruídos, recebendo e perdendo membros. Um processo pode juntar-se a um grupo e, no momento que bem entender, deixá-lo. Nada impede que um determinado processo pertença a vários grupos simultaneamente. Além do mais, quando uma mensagem é enviada para o grupo, todos os seus membros irão recebê-la. A seguir discutiremos as principais características e mecanismos oferecidos por protocolos de *group communication.*

4.2.1.1 Grupos Fechados x Grupos Abertos

Existem duas categorias de grupos de processos suportados pelos protocolos de *group communication*: grupos fechados e grupos abertos.

Nos grupos fechados, apenas os membros de um grupo podem enviar mensagens para o grupo inteiro. Processos externos ao grupo não podem fazê-lo, apesar de estarem habilitados a enviar mensagens individualmente para cada membro.

Ao contrário, os grupos abertos permitem que qualquer processo, seja ele membro ou não de um grupo, possa enviar mensagens que serão recebidas pelo grupo inteiro.

Grupos fechados são geralmente utilizados quando a característica do grupo é a de trabalho cooperativo. Um cliente, portanto, irá enviar uma mensagem requisitando um serviço a um dos membros do grupo individualmente e este, de maneira transparente ao cliente, irá interagir com o grupo de servidores do qual faz parte para atender ao pedido do cliente. Já os grupos abertos são utilizados quando seus membros são servidores replicados de um determinado serviço. Neste caso é importante que um processo não membro do grupo (um cliente), possa interagir com o grupo por inteiro.

4.2.1.2 Controle de Pertinência de Membros

O mecanismo de controle de pertinência *(group membership)* gerencia as operações de criação e destruição de grupos, bem como possibilita a adição e remoção de membros de grupos. Para isto, é mantida uma base de dados com informações sobre o grupo e seus processos membros. Uma das abordagens para o controle de pertinência é a utilização de um processo *group server*, que é responsável pela manutenção da base de dados do grupo e por processar as requisições de criação e remoção de grupos, inclusão e exclusão de membros nos grupos, etc. Esta é uma alternativa bastante simples de ser implementada. Infelizmente, por ser um serviço centralizado, possuindo um único ponto de falha, não é uma solução suficientemente robusta. Caso o *group server* falhe, o grupo estará automaticamente dissolvido.

Uma segunda abordagem é o controle de pertinência distribuído, ou seja, todos os membros do grupo mantêm bases de dados locais sobre o grupo. Para que isto seja possível, os processos que desejam fazer parte do grupo deverão enviar uma mensagem de *join* para o grupo inteiro. Com os grupos abertos, isto é perfeitamente possível. Já em grupos fechados, para que um processo externo junte-se ao grupo, deve haver a possibilidade deste enviar a mensagem de *join* para o grupo.

Quando um membro de um grupo falha, este efetivamente está fora do grupo. Porém, como este membro não deixou o grupo voluntariamente, fica a cargo dos membros remanescentes descobrirem isto. Após a constatação da falha de um dos membros do grupo, este será removido das bases de dados do grupo mantidas por cada membro individualmente. Caso um número significativo de membros do grupo falhe, deve haver mecanismos de reconstrução deste grupo. Além de excluir do grupo processos isolados que venham a falhar, esta característica cobrirá situações de particionamento de uma rede em duas ou mais sub-redes, onde, em cada uma delas surgirá um sub-grupo derivado do grupo de processos original.

A circulação de mensagens do grupo deverá ser sincronizada com o controle de pertinência, ou seja, a partir do momento em que um processo se juntar ao grupo, este passará a receber todas as mensagens enviadas ao grupo depois de sua adesão. Da mesma forma, a partir do momento em que um processo abandonar o grupo, não mais receberá qualquer mensagem destinada ao grupo.

4.2.1.3 Endereçamento

Para que um grupo possa receber mensagens de processos externos, é necessário que ele possua alguma maneira de ser endereçado. Em redes que suportam *multicast*, os membros de um grupo podem ser associados a um endereço *multicast*. Qualquer mensagem enviada para este endereço será recebida por todos os membros do grupo. Caso a rede em questão não suporte *multicast*, mas *broadcast*, uma mensagem por difusão é enviada para todas as suas máquinas. Cada núcleo que receber a mensagem deverá extrair desta o identificador do grupo e entregar a mensagem aos membros deste grupo que eventualmente estiverem executando naquela máquina.

Caso o sistema não suporte nem *multicast* nem *broadcast*, o núcleo do sistema operacional da máquina origem da mensagem deverá ter uma lista de máquinas que possuem processos membros do grupo. O núcleo, então, enviará a mensagem individualmente para cada uma delas. Note que, em qualquer uma das três abordagens, o processo cliente envia apenas uma mensagem para o grupo. Como a entrega será implementada, fica a cargo do serviço de *group communication*.

4.2.1.4 Atomicidade

Esta característica da comunicação em grupo de processos indica que quando uma mensagem é enviada a um grupo, ela ou é recebida por todos os seus membros ou por nenhum deles.

A atomicidade é importante para tornar mais fácil a programação distribuída. Um processo que enviou uma mensagem para um grupo não precisa se preocupar com relação a quais dos membros a mensagem foi recebida e quais não receberam a mensagem com sucesso. Desta forma, é garantido que todos os processos membros do grupo estão consistentes entre sí, ou seja, receberam as mesmas mensagens e, portando, têm condições de possuir o mesmo estado interno.

4.2.1.5 Ordem na Entrega das Mensagens

Juntamente com a atomicidade, esta é a característica mais forte da comunicação em grupos de processos. É garantido que todas as mensagens enviadas para um grupo de processos serão recebidas na mesma ordem por todos os seus membros.

Suponhamos que dois processos clientes (C1 e C2) estejam fazendo atualizações em uma base de dados mantida por um serviço de banco de dados. Este serviço é composto por um grupo de processos servidores de banco de dados, com o objetivo de oferecer tolerância a faltas através da replicação da base de dados. O cliente C1 enviou para o grupo a atualização A1. Imediatamente após, o cliente C2 envia a atualização A2. Todos os servidores receberão A1e A2, segundo a propriedade da atomicidade, apresentada anteriormente. Só que isto não é o

suficiente para que todas as bases de dados mantidas pelo grupo estejam consistentes. Supondo que todas as atualizações estejam sendo feitas no mesmo registro e em um mesmo campo, caso alguns membros do grupo considerado, por qualquer motivo que seja, recebam as mensagens em uma ordem diferente de outros membros (alguns servidores recebam as mensagens na sequência A1, A2; outros na seqüência A2, A1), poderá haver inconsistência entre as bases que deveriam ser réplicas uma da outra.

Para que isto não ocorra, os protocolos de *group communication* garantem que todo o grupo receba as atualizações na mesma ordem, seja ela qual for.

4.2.2 A Utilização de Comunicação em Grupo no Bouncer

O serviço de comunicação em grupos de processos irá ser utilizado para comunicar os vários *daemons* servidores Bouncer, distribuídos pelas diversas máquinas de uma rede. Como visto anteriormente, estes vários processos irão trabalhar de forma cooperativa, trocando informações através de um mecanismo de comunicação com uma série de características especiais, oferecidas pelos serviços de *group communication*. Vejamos como o Bouncer se utiliza de cada uma dessas características:

Grupos abertos x grupos fechados - O grupo Bouncer é fechado, uma vez que possui um perfil de trabalho cooperativo. Um cliente se comunica com o membro local do grupo, solicitando licença para executar e este membro local repassa, então, o pedido para o restante do grupo.

Controle de pertinência - Conforme especificado no protocolo Bouncer, quando um *daemon* B é inicializado, este procura juntar-se a um grupo Bouncer já existente ou criá-lo, caso o grupo não exista. Em caso de falha de algum dos *daemons* B, esta falha deverá ser detectada e o processo em questão removido do grupo Bouncer. Todas estas atividades irão fazer uso do controle de pertinência.

Endereçamento - O Bouncer não possui nenhum tipo de requisito especial quanto à forma com que o serviço de *group communication* irá distribuir mensagens enviadas a um grupo de processos entre seus membros.

Atomicidade - Esta característica é fundamental para o Bouncer, uma vez que cada *daemon* servidor mantém uma base de dados local (arquivo de licenças) contendo todas as licenças em uso na rede. Estas bases de dados devem estar consistentes em todos os nós da rede onde existir um processo B em execução, pois ela é quem determina o estado interno do grupo Bouncer. Portanto, todas as mensagens enviadas ao grupo Bouncer, como pedidos e liberações de licenças, deverão ser recebidas por todos os seus membros.

Ordem na entrega das mensagens - O Bouncer deve dispor de algum mecanismo que resolva condições de corrida em casos como aplicações disputando por uma única licença disponível. Pode ocorrer de várias aplicações

solicitarem licença em um intervalo de tempo bastante pequeno, ou até simultaneamente. Caso não existam licenças disponíveis para atender a todas estas requisições, o critério de desempate adotado será a ordem com que os daemons servidores receberem estas mensagens do grupo.

Além das características tradicionais existentes em ferramentas de comunicação em grupo de processos, já discutidas nas seções anteriores, a API de *group communication* a ser utilizada para a implementação do Bouncer deverá tratar reunificação de grupos.

Tendo em vista a situação descrita na seção que fala do serviço de pertinência, onde um grupo de processos pode ser fragmentado em vários sub-grupos quando a rede é particionada, faz-se necessário algum mecanismo que, uma vez solucionada esta segmentação, consiga promover um re-agrupamento entre os sub-grupos derivados, reconstruindo, assim, o grupo original. Passaremos a descrever um possível mecanismo para possibilitar que um grupo de processos B fragmentado possa ser reunificado.

Cada grupo de processos deverá possuir um membro com um *status* especial de líder do grupo. Um processo é eleito líder de grupo quando é o membro mais antigo do grupo ao qual pertence. Quando um processo junta-se a um determinado grupo, encontra-se embutida nesta operação a criação deste grupo, caso o mesmo ainda não exista. Ao ser o primeiro membro de um grupo, logicamente um processo é automaticamente seu líder. Caso um líder de grupo falhe, a liderança será herdada pelo membro ativo mais antigo do grupo.

Cada processo líder de grupo deverá, periodicamente, enviar uma mensagem por difusão afirmando sua liderança (chamaremos esta mensagem de **mensagem de liderança**), não sendo permitida a coexistência de vários líderes de grupo semelhantes em sua rede local. Dois líderes de grupo são semelhantes quando os serviços oferecidos por eles e por seus respectivos grupos são idênticos (por exemplo, dois líderes de grupos Bouncer são semelhantes). Caso seja detectada a presença de um ou mais líderes semelhantes em uma mesma rede, isto significa que a quantidade de segmentos correspondente ao número de líderes de grupo foi fisicamente reunificada e que, conseqüentemente, mais de um grupo semelhante passará a existir em uma mesma rede local. Estes sub-grupos, portanto, deverão ser aglutinados em um único grupo. Note que, no caso particular do Bouncer, foi comentado que a fragmentação do grupo Bouncer em n sub-grupos implicará na multiplicação por n do número de licenças disponíveis para uma determinada aplicação. Portanto, quando dois ou mais sub-grupos Bouncer passarem a coexistir em uma mesma rede, após a junção de várias sub-redes, pode ocorrer que um número de cópias de uma determinada aplicação protegida em execução seja bem maior que a quantidade de licenças para ela disponibilizada. Esta situação pode ser resolvida, especificamente para o Bouncer, de maneira trivial, devido à sua característica de re-inicialização automática.

Um processo B líder de grupo, ao receber uma mensagem de liderança de algum outro líder, imediatamente irá enviar uma mensagem *kill group* para o seu grupo, determinando que todos os membros deverão finalizar suas execuções, causando, assim, a extinção daquele grupo. Esta operação é repetida para vários sub-grupos, até que todos estes tenham sido extintos ou que apenas um deles sobreviva. Devemos lembrar que uma aplicação protegida pelo Bouncer verifica periodicamente a existência do servidor B. Caso seja constatado que este não está ativo, um novo pedido de licença de execução será feito pela aplicação cliente, o que causará a re-inicialização de B. Assim sendo, após a extinção de todos os sub-grupos Bouncer de uma rede e, conseqüentemente, todos os servidores B, ou quando restar apenas um deles, os processos B, ao serem re-inicializados por seus respectivos clientes, irão juntar-se ao grupo sobrevivente ou a um novo e único grupo que será formado.

Tomemos como exemplo uma rede R, contendo 5 máquinas, onde em cada uma delas encontra-se em execução um servidor B. Estes processos B estão compondo, portanto, um grupo de processos Bouncer, que chamaremos de RBouncer, com seu respectivo líder de grupo (BLíder). Uma segmentação de R causou o surgimento de R' e R" (veja a figura 4.2). Com isto, os membros de RBouncer do segmento R', detectando a falta de comunicação com os membros localizados em R", dão origem a um sub-grupo de RBouncer, ao qual chamaremos RBouncer'. De maneira análoga, um sub-grupo RBouncer" será redefinido em R" pelos processos B nela localizados. RBouncer' e RBouncer" têm como líderes de grupo BLíder' e BLíder", respectivamente. Uma determinada aplicação protegida pelo Bouncer (C, por exemplo), com 10 licenças de uso concorrente, após a segmentação da rede, passará a ter 20 licenças disponíveis (10 por segmento). O cenário acima descrito encontra-se representado na figura 4.2.

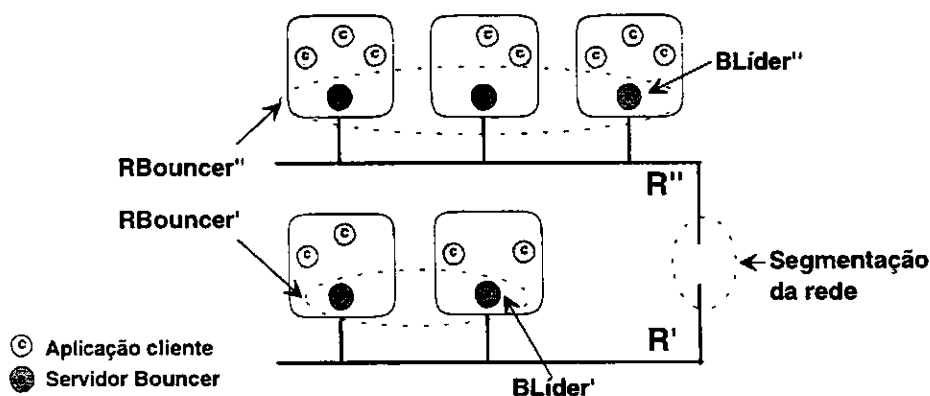


Figura 4.2 - Formação de sub-grupos Bouncer causada por segmentação da rede

Em cada um dos segmentos, os líderes BLíder' e BLíder", periodicamente, enviam *broadcasts* com suas mensagens de liderança. Considere a seguinte situação: Existem 5 cópias de C em execução no segmento R' e 8 no segmento R".

Após a re-conexão destes segmentos, BLíder" difunde sua mensagem de liderança. Ao receber tal mensagem, BLíder' extingue o grupo RBouncer' através do envio de uma mensagem *kill group* para o grupo. Os processos B, ex-membros de RBouncer' serão re-inicializados pelos seus respectivos clientes e, à medida que forem entrando em execução, juntar-se-ão ao grupo sobrevivente RBouncer". Note que neste momento, como no grupo sobrevivente haviam 8 licenças de C em uso, restaram apenas 2 licenças para serem disputadas pelas aplicações clientes que executavam em R'. Assim, das 5 que antes estavam em execução, 2 delas conseguirão prosseguir executando, enquanto as demais terão seus pedidos de licença negados. A situaçao em que BLíder' difunde sua mensagem de liderança primeiro, causando a extinção do grupo RBouncer" é análoga.

Uma outra situação possível seria, após re-conectadas R' e R", simultaneamente ambos os líderes de grupo enviarem suas respectivas mensagens de liderança. Neste caso, BLíder' e BLíder" iriam receber tais mensagens e extinguir seus grupos, não havendo grupo sobrevivente. O grupo seria totalmente reconstruído e as aplicações iriam disputar todas as 10 licenças agora disponíveis.

É importante perceber que, no caso da existência de um número maior de segmentos na rede sendo unificados, a sequência de passos descrita acima poderá ser repetida várias vezes até que se tenha finalmente um único grupo Bouncer na rede.

Note que o suporte de IPC para o envio de uma mensagem de liderança é diferente do utilizado para o envio de uma mensagem *kill group*. As mensagens de liderança são enviadas pelos líderes de grupo através de um *broadcast* de rede, onde todos os processos da rede podem potencialmente recebê-la. A mensagem *kill group*, por sua vez, utiliza o suporte de entrega de mensagens do protocolo de *group communication*, pois a recepção de tal mensagem deve estar restrita aos membros do sub-grupo Bouncer jurisdicionado pelo líder de grupo que a enviou.

Um aspecto importante a ser discutido aqui é a freqüência com que as mensagens de liderança deverão ser enviadas por um líder de grupo. Esta freqüência será determinada por um *time out* adaptativo, ao qual chamaremos ***time out* de liderança (LTOut)**, cujo valor é inversamente proporcional à freqüência com que ocorre particionamento na rede. Portanto, quanto maior a incidência de particionamento, mais freqüentemente um líder de grupo irá procurar pela existência de outros líderes para que sejam feitas as devidas re-unificações de eventuais sub-grupos Bouncer coexistentes na mesma rede.

O valor do *time out* de liderança deverá variar dentro de um intervalo fechado, limitado pelas constantes **LTOutMin** e **LTOutMax**. Este intervalo é importante para que o tempo entre o envio de mensagens de liderança não seja tão pequeno a ponto de causar tráfego excessivo na rede ($LTOut \geq LTOutMin$) nem tão grande que impossibilite a detecção de uma reunificação de segmentos de rede em um tempo razoável ($LTOut \leq LTOutMax$). Portanto, o cálculo de LTOut será dado por:

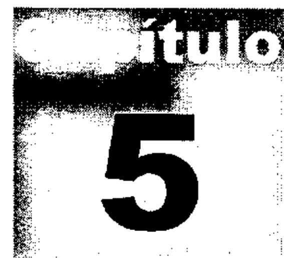
LTOut = max(LTOutMin, min(LTOutMax, 1/fkm))

onde:

LTOut	Intervalo de tempo (em segundos) entre o envio de duas mensagens de liderança.
LTOutMin	Limite mínimo para LTOut.
LTOutMax	Limite máximo para LTOut.
max(x, y)	Função que seleciona o argumento de maior valor numérico.
min(x, y)	Função que seleciona o argumento de menor valor numérico.
fkm	Freqüência de envio de mensagens *kill group* dentro de um grupo.

Observe que um particionamento de rede só é detectada no momento em que ela é desfeita, ou seja, quando um líder de grupo percebe a existência de, pelo menos, outro líder semelhante a ele e extingue seu grupo através do envio da mensagem *kill group*. Em última instância, a freqüência com que esta mensagem é enviada irá refletir a freqüência de particionamento da rede.

Como último requisito de comunicação para a implementação do Bouncer, devemos dispor de uma primitiva de *broadcast*, para que esta possa ser utilizada no processo de recomposição dos grupos (envio das mensagens de liderança). Conforme discutido no tópico referente à forma de endereçamento utilizada por grupos de processos, não é nossa preocupação a maneira como esta primitiva de broadcast é implementada (utilizando *broadcast* de rede realmente ou vários *unicasts*), contanto que o Bouncer consiga atingir todas as máquinas da rede com uma única chamada a esta primitiva de difusão.



5 Conclusão

Este trabalho discutiu a necessidade que a indústria de software tem em relação à utilização de mecanismos que protejam seus produtos contra o uso não autorizado, em face dos prejuízos causados ao setor por esta prática. Foram apresentadas várias das alternativas disponíveis para proteger a propriedade intelectual dos desenvolvedores, sendo estas classificadas em proteção legal, proteção por hardware e proteção por software. Analisando cada uma destas soluções, no entanto, foi constatado que uma série de inconvenientes pode comprometer a eficácia da respectiva proteção oferecida. A proteção legal necessita de elementos de fiscalização, que garantam sua aplicação. A proteção por hardware, apesar de bastante eficiente e livre de suporte, limita a flexibilidade no uso do software protegido, além de manter uma alta dependência das constantes evoluções do hardware. As alternativas de proteção por software, ou são burladas com relativa facilidade, ou necessitam da manutenção de uma infra-estrutura por parte do desenvolvedor para a geração de chaves de ativação e interação com os usuários finais, ou ainda geram atividades adicionais de suporte técnico/administrativo.

Dentro deste universo, os serviços de gerenciamento de licenças, que são soluções por software para a coibição da pirataria, apresentam-se como um paradigma que vem ganhando bastante força e popularidade entre desenvolvedores e administradores. Vimos que esta, além de ser a solução técnica mais flexível para a proteção de software, pode atuar como uma importante ferramenta nas estratégias de *marketing* e distribuição dos produtos de software [Élan 95].

No capítulo 2 foi descrito o funcionamento do serviço de gerenciamento de licenças, que é um sistema distribuído que controla a utilização de produtos de software segundo vários critérios para limitar o acesso de um usuário a uma aplicação. Estes critérios foram tratados ao longo deste trabalho como políticas de licenciamento ou de proteção de software. Tivemos oportunidade de analisar alguns dos mais importantes gerenciadores de licenças existentes atualmente no mercado.

Foi visto que o serviço de gerenciamento de licenças oferece uma grande flexibilidade para que os desenvolvedores de software implementem e mantenham suas políticas de licenciamento. Toda esta flexibilidade, no entanto, tem como contrapartida a geração de uma série de atividades de configuração e suporte para o administrador do sistema/rede do ambiente computacional onde irão ser executadas as aplicações protegidas.

Como forma de contornar estas limitações, apresentamos no capítulo 3 deste trabalho o Bouncer, a nossa solução para o gerenciamento de licenças, que se propõe a solucionar as deficiências apontadas nos demais gerenciadores. Além de proteger as aplicações contra uso ilegal de forma confiável e tolerante a faltas, o Bouncer veio eliminar os problemas de suporte administrativo comuns a ferramentas deste tipo.

O Bouncer tem compromisso com a simplicidade e a robustez, visando ser uma alternativa viável para os desenvolvedores de pequeno porte que atuam no mercado de software de prateleira. A simplicidade do Bouncer limita sua flexibilidade com relação a gerenciadores de licença mais sofisticados. Esta limitação de flexibilidade está no fato de que nem todas as modalidades de políticas de licenciamento são oferecidas, além do que, não é possível converter cópias de demonstração de programas em cópias definitivas através da utilização do Bouncer. Foi visto, no entanto, que estas características não oferecidas pelo Bouncer não são limitantes significativos para o segmento do mercado de software a que este busca atender, visto que a principal política de licenciamento utilizada por aplicações de rede é a de quantidade de ativações concorrentes, que é oferecida pelo Bouncer. Além do mais, não é uma prática comum no mercado de software de prateleira a conversão de cópias de demonstração em cópias definitivas, mediante o fornecimento de chaves de ativação ou de autenticação de parâmetros de licenciamento. Esta prática demanda a alocação de pessoal de suporte ao cliente, o que não é interessante para pequenos desenvolvedores, que possuem estruturas modestas de pessoal. A robustez, que é um requisito que todos os serviços de licenciamento buscam oferecer, é um dos pontos fortemente trabalhados no Bouncer, através de sua arquitetura e de seu protocolo de funcionamento.

A arquitetura do Bouncer possui uma peculiaridade não encontrada em nenhum outro serviço de gerenciamento de licenças, que é a hibridez entre os modelos de programação cliente-servidor e *peer-to-peer*. No âmbito de uma máquina da rede, um, e apenas um servidor encontra-se ativo, atendendo a vários clientes. Já sob o ponto de vista da rede como um todo, cada ponto da rede que possui uma aplicação licenciada em execução possui um servidor ativo, e os servidores das diversas máquinas trabalham de maneira cooperativa. Desta forma, os clientes não precisam de nenhum serviço adicional para determinar em que máquina da rede está o servidor (este estará ativo em todas as máquinas onde existam aplicações protegidas sendo executadas). O fato de que todos os pontos da rede podem potencialmente possuir um servidor ativo tira do administrador a

carga de trabalho adicional de ter que configurar a localização do serviço (possivelmente em várias máquinas, se for desejado replicá-lo para prover tolerância a faltas). No Bouncer, a tolerância a faltas através de replicação passa a ser uma característica intrínseca à sua arquitetura.

No capítulo 5 foram estudados com detalhes os serviços de comunicação entre processos utilizados na especificação do Bouncer. Para que as aplicações protegidas pelo serviço de licenciamento Bouncer se comuniquem com os respectivos *daemons* servidores, é utilizado o modelo cliente-servidor, através de chamadas a procedimentos remotos, ou RPC. Foi visto que, como esta comunicação entre cliente e servidor é feita a nível local, sem a utilização da rede, o serviço de RPC a ser utilizado pelo Bouncer pode ser bastante simples. Já na comunicação entre os vários *daemons* servidores Bouncer, uma vez que estes constituem um grupo de processos trabalhando de maneira cooperativa, foi especificado o uso de serviços de comunicação entre grupos de processos, ou *group communication*, desde que estes possuam a característica especial de tratamento de particionamento de redes.

5.1 Contribuições do Trabalho

A principal contribuição dada por este trabalho é a especificação de um serviço de gerenciamento de licenças de software que não gere atividades adicionais de suporte técnico e administrativo. Especificamos um serviço que, uma vez instalado em uma máquina, dispensa qualquer tipo de atenção do administrador do sistema/rede, seja para configurar políticas de licenciamento e localização de servidores, seja para re-inicializar o serviço após a ocorrência de falhas. Como podemos observar na tabela abaixo, que lista os principais problemas de suporte detectados em serviços de gerenciamento de licenças, estes problemas não são encontrados em nossa solução de licenciamento.

Atividade de suporte para	FlexLM	ÉlanLM	iFOR/LS	Bouncer
Reinicialização manual do servidor de licenças	SIM	NÃO	SIM	NÃO
Manipulação do arquivo de licenças	SIM	SIM	SIM	NÃO
Configuração da localização dos servidores replicados	SIM	SIM	SIM	NÃO
Liberação de licenças pendentes manualmente	SIM	NÃO	SIM	NÃO
Limitação no número de réplicas do servidor	SIM	NÃO	NÃO	NÃO
Interação com desenvolvedores para cálculo de chaves de licenças e/ou ativação do módulo servidor	SIM	SIM	SIM	NÃO

Tabela 5.1 - Quadro comparativo entre o Bouncer e outros gerenciadores de licença sobre problemas de suporte.

Como apresentado em várias partes deste trabalho, o *daemon* servidor Bouncer é automaticamente re-inicializado após a detecção de sua falha. Vimos que isto é feito pelas aplicações por ele protegidas. O Bouncer não utiliza qualquer arquivo para armazenar informações sobre licenças em uso ou políticas de licenciamento. Estas informações encontram-se em memória. Como a disposição dos *daemons* servidores Bouncer é não hierárquica, baseada no modelo *peer-to-peer*, isto significa que em cada máquina onde houver uma aplicação protegida em execução, haverá necessariamente um daemon Bouncer ativo. Isto elimina a necessidade de se especificar a localização do(s) servidor(es), além de não impor nenhuma limitação quanto ao número de servidores ativos. Como visto na apresentação do protocolo Bouncer, no capítulo 3, as licenças pendentes ou órfãs não precisam ser liberadas através da intervensão humana, uma vez que o processo de auditoria fará isto de forma automática. Em nenhum momento do funcionamento do Bouncer é requisitada qualquer chave de ativação, não havendo a necessidade de interação entre compradores e desenvòlvedores para a obtenção deste tipo de informação.

Além do aspecto de eliminação de suporte, nosso serviço de gerenciamento de licenças pode apresentar vantagens sobre seus similares no que diz respeito ao custo de aquisição. No capítulo 2, foram comentados os preços dos principais gerenciadores de licença do mercado, incluindo algumas políticas de preços praticadas por seus fornecedores. O FlexLM custa US$ 40,000, com direito a produzir um número ilimitado de licenças para seus compradores. A GLOBEtrotter oferece também uma política de preços direcionada para pequenas empresas em ascensão. Esta política especial consiste em cotar o FlexLM de acordo com as vendas anuais do desenvolvedor (*pay as you grow*) [Miller 96]. O kit de desenvolvimento do ÉlanLM custa em torno de $7,500 para a primeira plataforma adquirida, diminuindo a cada nova plataforma [Lovett 96]. O preço, por kit de desenvolvimento, do iFOR/LS é de US$ 8,000, devendo o desenvolvedor pagar ainda uma anuidade de US$ 7,500 pelos direitos de criação de um número ilimitado de licenças para os compradores [Leveille 96]. O sentimento que temos sobre o Bouncer com relação ao seu custo de aquisição é de que, por se tratar de uma ferramenta bastante simples, pode ser adquirido por um preço bastante acessível para os pequenos desenvolvedores. Esta avaliação foi feita a partir da implementação de um protótipo do Bouncer (comentado a seguir), onde pudemos ter uma idéia do esforço de programação exigido para implementá-lo.

5.2 Sugestões para Trabalhos Futuros

Como sugestões para trabalhos que venham a dar continuidade ao projeto Bouncer, temos:

1. Implementar o Bouncer de acordo com as especificações feitas neste trabalho, utilizando serviços de comunicação otimizados para o tipo de requisitos necessários ao funcionamento do Bouncer.

Devido à indisponibilidade de ferramentas de *group communication* durante a fase de produção do primeiro protótipo do Bouncer, este foi implementado apenas com a utilização de RPC, de tal forma que tanto a realização de comunicação ponto-a-ponto (troca de mensagens entre aplicação e *daemon* B) quanto comunicação por difusão (troca de mensagens entre o grupo de *daemons* B) fizeram uso deste paradigma. Para que os servidores B pudessem se comunicar, algumas adaptações tiveram que ser feitas na função de *broadcast* disponível na API de RPC por nós utilizada (Clnt_broadcast()), já que esta não possui uma semântica adequada para as nossas necessidades. Com a atual disponibilidade de ferramentas de GC em nosso ambiente computacional, uma próxima etapa será o desenvolvimento de um novo protótipo do Bouncer, utilizando um RPC simplificado e ferramentas de GC que atendam a todos os requisitos exigidos pelo protocolo Bouncer, especificados no capítulo anterior. Atualmente, alunos desenvolvendo trabalhos de iniciação científica junto ao Laboratório de Sistemas Distribuídos estão trabalhando com alguns serviços de comunicação em grupo de processos, como o Horus [Horus 96] e o Ensemble [Ensemble 96], visando utilizá-las na implementação do novo protótipo do Bouncer.

2. Acrescentar à especificação do Bouncer, suporte a outras modalidades de licenciamento.

Com o intuito de tornar o Bouncer o mais simples possível, algumas modalidades de licenciamento não foram especificadas neste trabalho. Em futuras versões do Bouncer, caso se deseje adaptá-lo para atender a um mercado de software mais sofisticado, podem ser incorporadas modalidades de licenciamento que permitam a produção de cópias de demonstração de aplicações, a possibilidade de converter estas cópias para cópias definitivas, além de também darem suporte à ampliação do número de licenças concorrentes disponíveis para uma determinada aplicação protegida pelo Bouncer. Estas modificações, no entanto, iriam gerar algumas atividades de suporte técnico para Bouncer, como, por exemplo, a necessidade da existência de um arquivo de descrição das políticas de licenciamento, a consequente manipulação deste arquivo, além da necessidade de interação com o desenvolvedor da aplicação para a configuração das políticas de licenciamento. A tabela a seguir mostra um novo quadro comparativo entre o Bouncer e os gerenciadores de licenças apresentados neste trabalho, considerando as modificações propostas acima.

Atividade de suporte para	FlexLM	ÉlanLM	iFOR/LS	Bouncer
Reinicialização manual do servidor de licenças	SIM	NÃO	SIM	NÃO
Manipulação do arquivo de licenças	SIM	SIM	SIM	SIM
Configuração da localização dos servidores replicados	SIM	SIM	SIM	NÃO
Liberação de licenças pendentes manualmente	SIM	NÃO	SIM	NÃO
Limitação no número de réplicas do servidor	SIM	NÃO	NÃO	NÃO
Interação com desenvolvedores para cálculo de chaves de licenças e/ou ativação do módulo servidor	SIM	SIM	SIM	SIM

Tabela 5.2 - Quadro comparativo entre o Bouncer modificado e outros gerenciadores de licença sobre problemas de suporte.

Note que mesmo com o surgimento de tais problemas de suporte, observamos que o Bouncer ainda apresenta menos problemas de suporte que os demais gerenciadores.

Referências Bibliográficas

[Birrell 84] BIRRELL, A.D. and NELSON, B.J. *Implementing Remote Procedure Calls*, ACM Trans. on Computer Systems, vol. 2, pp. 39-59, February, 1984.

[Birman 91] BIRMAN, K., SHIPER, A. and STEPHENSON, P. *Lightweight Causal and Atomic Group Multicast*, ACM Transactions On Computer Systems, Vol. 9, No. 3, pp. 272-314, August, 1991.

[Drummond 96] DRUMMOND, Rogério e HOYOS, Carlos. *Linear - Linearizador de Estruturas Complexas*, White-paper, Laboratório A-HAND, Campinas-SP, maio, 1996.

[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.

[Ensemble 96] Horus Project. *Introduction to Ensemble*, http://www.cs.connell.edu/Info/Projects/Ensemble/Overview.html, December, 1996.

[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.

[Greguras 94] GREGURAS, Fred M., WONG, Sandy Jane. *Software Licencing Flexibility Complements the Digital Age*, http://www.catalog.com/napmssv, December, 1994.

[Hoare 78] HOARE, C.A.R., *Communicating, Sequential Processes*, Communications of the ACM, Vol. 21, No. 8, pp. 666-677, August, 1978.

[Horus 96] Horus Project home-page, http://www.cs.connell.edu/Info/Projects/Horus, December, 1996.

[Laprie 89] LAPRIE, J-C., *Dependability: a Unifying Concept for Reliable Computing and Fault Tolerance*, Dependability of Resilient Computers, 1989.

[Lemos-Veríss 91] LEMOS, R. e VERÍSSIMO, P., *Confiança no Funcionamento - Resposta para uma Terminologia em Português*, Comunicação pessoal, dezembro de 1991.

[Lotus 90] Lotus Inc., *Lotus 123 para Unix - Manual do Usuário*, abril, 1990.

[OpenView 96] Hewlett Packard Inc. *OpenView Home Page*, http://hpcc998.external.hp.com:80/nsmd/ov/main.html, December 1996.

[IBM 96] IBM Inc. *iFOR/LS and NCS*, http://www.raleigh.ibm.com/ifs/ifsncs.html, 1996.

[Intel 96] Intel home-page, http://www.intel.com, December, 1996.

[Leveille 96] LEVEILLE, Greg. Gradient Technologies, Inc. Technical Support, Comunicação pessoal, dezembro, 1996.

[Lovett 96] LOVETT, Debborah. Élan Computer Group Technical Support, Comunicação pessoal, dezembro, 1996.

[Macedo 95] MACÊDO, Raimundo J. A., EZHILCHELVAN, P., SHRIVASTAVA, Santosh K., *Flow Control Schemes for a Fault-Tolerant Multicast Protocols*, The proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems, December, 1995.

[Macedo 95b] MACÊDO, Raimundo J. A., SHRIVASTAVA, Santosh K., *The Implementation and Performance Analysis of a Total Order Delivery Protocol for Group Communication*, The Proceedings of XXI Latin American Conference of Informatics and the XV Congress of the Brazilian Computer Society, pp. 287-299, July, 1995.

[Miller 96] MILLER, Susan. GLOBEtrotter Software, Inc. Technical Support, Comunicação pessoal, dezembro, 1996.

[Remer 87] REMER, Daniel, ELIAS, Stephen. *Legal Care for Your Software - A Step by Step Guide for Computer Software Rights and Publishers*, 3th Edition, September, 1987.

[Sauvé 96] SAUVÉ, Jacques P. Consultor da Light-Infocon Tecnologia S.A., Comunicação pessoal, novembro, 1996.

[Schneier 93] SCHNEIER, Bruce. *The IDEA Encryption Algorithm*, Dr. Dobb's Journal, December, 1993.

[SCO 95] Santa Cruz Operation, Inc. *Network Programmer's Guide and Reference - chapter 7*, http://frodo.dc.luth.se:457/netguide/disockN.inet_sockets_intro.html, 1995.

[Sun 88] Sun Microsystems. *RPC: Remote Procedure Call Protocol Specification Version 2*, RFC1057 Internet Request For Comments, June 1988.

[Sun 96] Sun Added Value Reseller home-page, http://www.apunix.com/products/sun/sun_page.html, December, 1996.

[SunOS 90] Sun MicroSystems Inc. *SunOS Reference Manual*, pg. 54-59, March 1990.

[Tanembaum 89] TANEMBAUM, Andrew S. *Computer Networks*, pg. 14-19, Prentice-Hall, 1989.

[Tanembaum 92a] TANEMBAUM, Andrew S. KAASHOEK, M. F., *Eficient Reliable Group Communication for Distributed Systems*, Vrije Universiteit, Amsterdam, The Netherlands, 1992.

[Tanembaum 92b] TANEMBAUM, Andrew S. *Modern Operating Systems*, Prentice-Hall, 1992.

Apêndice A

Código Fonte do Monitor

```
/*
 * monitor.c
 * "Casca" para monitorar licencas utilizadas em todo o dominio local da rede
 */
#define MAXNODES        1000        /* numero maximo de maquinas na rede */
#include <stdio.h>
#include <string.h>
#include <sys/errno.h>
#include <bapi.h>
#include <error.h>

/*
 * Globais
 */
char            bufstdout[ BUFSIZ ];    /* para buferizacao da saida padrao */
extern int      errno;                  /* codigo de erro retornado por B_Monit*/
main( argc,argv )
int     argc;
char    *argv[]; {

        char            *p;
        char            *prodt;
        char            *host;
        int             vetlen;
        int             tout;
        int             logon;
        int             loglevel;
        Config          conf;
        ListNodes       LList;          /* retorno da funcao B_Monit() */

/*
 * buferiza para saida padrao
 */
```

```
        setbuf( stdout, bufstdout );
        prodt = NULL;
        host = NULL;
        vetlen = MAXNODES;

/*
 * obtem parametros para monitoramento
 */
while ( argc > 1 && *argv[ 1 ] == '-' ) {
                p = argv[ 1 ] + 1;
                while ( *p ) {
                        switch ( *p ) {
                                case 'a': /* nome da aplicacao  monitorada */
                                        p++;
                                        prodt = p;
                                        goto prox_arg;
                                        break;
                                case 'h': /* host a ser monitorado */
                                        p++;
                                        host = p;
                                        goto prox_arg;
                                        break;
                                default:
                                 sintax( argv[ 0 ]);
                                 break;
                        }
                        ++p; /* processa proximo caracter da opcao */
                }
 prox_arg:
        argc--; argv++;
        }

       /*
        * chama funcao de monitoramento
        */
        if( (LList = B_Monit( prodt, host, &vetlen )) == NULL ){
                fprintf( stderr, "\n%s: ocorreu erro fatal %d\n", argv[0], errno);
```

```
                exit( 1 );
        }
        m_Display( LList, vetlen);
        exit( 0 );
}

/*
 * Display
 * objetivo:
 *      exibir vetor com licencas ativas de produtos em uma  maquina da rede
 * retorna
 *      0, em caso de sucesso
 *      nao trata erros
 */
Display( Lista, tam )
Prodtnode       *Lista;
int             tam;
{
        int     i;

        for( i = 0; i < tam; i++ ){
            printf("%s\t\t\t%d\t\t\t%d\n", Lista[i].ProdtId, Lista[i].MaxLicense,
                                                    Lista[i].ActLicense );
        }
        return( 0 );
}

/*
 * m_Display
 * objetivo:
 *    exibir dados sobre aplicacoes policiadas executando em maquinas da rede
 * retorna:
 *      0, em caso de sucesso
 *      nao trata erros
 */
m_Display( LList, VetLen )
ListNodes       LList;
```

```
int     Len;
{
        int     i;
        printf("\n\t\t\t\tM O N I T O R\n");
        printf("****************************************************************
*************\n");
        for( i = 0; i < VetLen; i++ ){
                printf("\nHost:\t%s\n\n", LList[ Len ].IPAddr);
                printf("Produto\t\tMaximo de Copias\tCopias Ativas\n");
                printf("--------------------------------------------------------------------\n");
                Display( LList[ Len ].VetProd, LList[ Len ].VetLen);
        }
        printf("\n\n");
        return( 0 );
}

/*
 * Sintaxe do Monitor
 */
sintax( prog )
char    *prog;
{
    printf("Sintaxe: %s [-a<prodtid>] [-h<hostname>]\n", prog );
}
```