UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Avaliação de Abordagens para Captura de Informações da Aplicação

por

ADRIANO BRUM FONTOURA

Dissertação de mestrado submetida à avaliação,
Como requisito parcial, para obtenção do grau de
Mestre em Ciência da Computação

Profa. Dra. Ingrid Jansch-Pôrto
Orientadora

Porto Alegre, fevereiro de 2002.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Fontoura, Adriano Brum
Avaliação de Abordagens para Captura de Informações da Aplicação / por Adriano Brum Fontoura. - Porto Alegre: PPGC da UFRGS, 2002.
128 p. : il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2002. Orientadora: Jansch-Pôrto, Ingrid.

1. Captura de mensagens. 2. Linux. 3. Interceptação. 4. Transparência. 5. Sistemas distribuídos. I. Jansch-Pôrto, Ingrid. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profa. Wrana Panizzi
Pró-Reitor de Ensino: José Carlos Ferraz Hennemann
Pró-Reitor Adjunto de Pós-Graduação: Prof. Jaime Evaldo Fensterseifer
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

Agradecimentos

Agradeço a Deus, por esta oportunidade de vida, e por colocar pessoas tão especiais em meu caminho.

Agradeço a minha orientadora, Profa. Dra. Ingrid E. S. Jansch Pôrto, por sua dedicação, paciência, e confiança no trabalho que estávamos realizando.

Agradeço aos colegas do Grupo de Tolerância a falhas e aos professores, pelo apoio e a troca de idéias, e em especial ao Sérgio Luis Cechin, que além do suporte intelectual foi um amigo que soube dizer as palavras certas em momentos difíceis e decisivos.

Agradeço a URI Campus Santiago, na pessoa da Prof. Ayda Bochi Brum, diretora geral deste Campus, e demais diretoras, que acreditaram em minha capacidade de trabalho, e aos colegas de trabalho, pelo apoio e compreensão recebidos.

Agradeço a Lisandra pelo amor, o apoio e principalmente pela paciência, aos meus pais, irmãos e amigos, que souberam entender os motivos pelos quais muitas vezes tive que estar ausente.

E, por fim, agradeço a todos que me ajudaram a tornar este trabalho uma realidade.

Sumário

Lista de Abreviaturas

ATM	Asynchronous Transfer Mode
BOA	Basic Object Adapter
BSD	Berkeley Software Development
CORBA	Common Object Request Broker
CPU	Central Process Unit
DII	Dynamic Invocaton Interface
EOM	Electra Object Model
GPL	General Public License
GPL	General Public License
IBCS2	Intel Binary Compatibility Specification
IDL	Interface Definition Language
IIOP	Internet Inter-Orb Protocol
IP	Internet Protocol
MOP	Metaobject Protocols
OGS	Object Group Service
ORB	Object Request Broker Interface
POSIX	Portable Operating System Interface for UNIX
SIC	System Call Interposition
SII	Static Invocation Interface
SO	Sistema Operacional
SVR4	System V Release 4
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VM	máquina virtual

Lista de Figuras

Lista de Tabelas

Resumo

Numerosas pesquisas estão introduzindo o conceito de grupo em padrões abertos para programação distribuída. Nestas, o suporte a grupo de objetos por meio de *middlewares*, apresentam diferentes abordagens de interligação com a aplicação. Segundo princípios defendidos na tese de Felber, essas abordagens vão ao encontro do objetivo de facilitar o desenvolvimento e proporcionar confiabilidade e desempenho.

Neste contexto, localizou-se três enfoques básicos para a interligação com a aplicação, denominados integração, serviço, e interceptação, que utilizam a captura de mensagens para obtenção de informações ou como meio para adicionar novas funcionalidades às aplicações. A utilização dessas informações pode auxiliar no ajuste de parâmetros funcionais de serviços relacionados, na escolha de mecanismos, influindo em aspectos como, desempenho e segurança. Ao longo do estudo dessas abordagens, sentiu-se a necessidade de estudar detalhes e testar aspectos de implementação, suas premissas de uso e as conseqüências advindas da incorporação de seus mecanismos junto à aplicação.

Este trabalho visa apresentar uma análise do comportamento das referidas abordagens por meio da implementação de protótipos, possibilitando assim, investigar problemas relacionados ao emprego da técnica e suas conseqüências quando integradas à aplicação. Os objetivos específicos reúnem a busca de informações qualitativas, tais como: modularidade, transparência, facilidade de uso e portabilidade; e informações quantitativas, fundamentalmente traduzidas pelo grau de interferência no desempenho da aplicação. O desenvolvimento dos protótipos teve como início a busca por um ambiente que ofereceria suporte as condições necessárias para a implementação das diferentes abordagens. Percebeu-se que definir os mecanismos diretamente sobre uma linguagem de programação, como C ou C++, não era viável. As versões padrões dessas linguagens não oferecem mecanismos capazes de suportar algumas características de implementação como, por exemplo, a captura de mensagens na abordagem de interceptação. A possibilidade é introduzida apenas por extensões dessas linguagens.

Assim, a investigação de um ambiente de implementação voltou-se para mecanismos disponíveis em sistemas operacionais. A opção pela utilização do Linux visou atender alguns requisitos importantes para o desenvolvimento dos protótipos tais como: facilidade de instalação, boa documentação e código aberto. Este último é um ponto essencial, pois a construção de parte dos protótipos explora a programação em nível do sistema operacional. A linguagem de programação C foi escolhida como base para a implementação, já que as diferentes abordagens exploram tanto o nível do *kernel* como o nível do usuário, e é compatível com o Linux.

A etapa de desenvolvimento dos protótipos possibilitou a coleta de informações sobre aspectos qualitativos. As demais informações que fazem parte do perfil levantado por este trabalho sobre as abordagens, foram obtidas através da utilização dos protótipos em experimentos com duas aplicações distribuídas denominadas de “Ping-Pong” e “Escolha de Líderes”, que têm como característica geral à troca de mensagens, utilizando comunicação através de *sockets*. A realização de medidas em múltiplas execuções, avaliadas após o tratamento estatístico necessário, permitiu definir um perfil das diferentes abordagens.

Palavras-chave: captura de mensagens, Linux, interceptação, transparência, sistemas distribuídos.

TITLE: “EVALUATING APPROACHES OF THE CAPTURING OF APPLICATION INFORMATION”

Abstract

Some researchers have introduced a concept of group in open standards for distributed programming. To provide group support, they use middleware. According to Felber’s thesis, such different approaches should allow easier development, offer reliable applications and have small impact on performance.

Within this context, we have identified three basic approaches to get information from the applications: integration, service and interception. They explore the capturing of messages to get information from the applications or as means to add new functionalities to them. The appropriate use of the obtained information can help adjust the related-service functional parameters and choose appropriate mechanisms, which impacts on performance and security. This work analyzes how these approaches behave. From their detailed study and in view of comparing their use conditions, we realized that we should: stress some details related to their implementation; test practical conditions over real implementations; verify use conditions; and evaluate the consequences of attaching new mechanisms to the applications.

We have implemented prototypes of these approaches, which allowed the investigation of problems related to the use of these techniques and the consequences of attaching them to an application. The specific goals join the search for qualitative information, such as modularity, transparency, easiness to use and portability, and for quantitative information, basically evaluated by the degree of interference on the performance of the application. Prior to the development of the prototypes, we had to define an adequate environment that would offer all the necessary conditions to implementing the various approaches. The standard versions of most programming languages, such as C and C++, have shown to be inadequate because they do not offer the mechanisms needed to support some implementation features such as the capturing of messages used by the interception approach, for instance. This possibility was introduced later by some extensions proposed to these languages.

Consequently, we have investigated the available mechanisms on some operating systems. Our choice, Linux, is due to some fundamental requirements: it is easy to install, there is good documentation, it is open source. The latter was considered essential, because the development of some prototypes explores the operating system level. C language was chosen as a basis for the implementation because the various approaches explore both the kernel and user levels; this language includes the features that are necessary to support either level and is compatible with Linux.

The development stage of the prototypes made collecting qualitative information possible. All additional information about the approaches, which is part of the profile defined by this work, has been taken from practical experiments using prototypes of two distributed applications called “Ping-Pong” and “Leader Election”. Both explore a message passing mechanism by using communication through sockets. Multiple executions have been run, followed by the necessary statistical handling, which permitted the evaluation pf the approaches and definition of their profile.

Keywords: capturing of messages, Linux, interception, transparence, distributed systems.

1 Introdução

Os avanços tecnológicos na área da computação trouxeram muitas facilidades, por um lado, e o aumento da complexidade, por outro lado. Essa complexidade decorre principalmente das exigências crescentes dos usuários. Conseqüentemente, isto gerou um desafio para os desenvolvedores de *software,* que precisam tornar as aplicações mais eficientes, mas mantendo a complexidade do desenvolvimento em patamares aceitáveis.

Na busca de aplicações mais eficientes, as informações dos diferentes níveis do ambiente da aplicação são dados importantes a considerar. As informações obtidas dão indicativos sobre o comportamento das aplicações, através de parâmetros como: a quantidade e tipos de mensagens, fluxo de informações e dependências. O uso adequado dessas informações pode auxiliar no ajuste de parâmetros funcionais de serviços relacionados, na escolha de mecanismos e, por conseqüência, traz reflexos sobre aspectos tais como, desempenho e segurança.

Um dos trabalhos que contribuiu nas idéias iniciais relacionadas à motivação desta dissertação foi um artigo de Brzezinsky *et al*. [BRZ 95], que aborda especificamente a área de recuperação em sistemas distribuídos. Eles propuseram uma interface em que a aplicação classifica as mensagens segundo a possibilidade de se tornarem órfãs ou perdidas, sem que isso venha acarretar modificações às próprias mensagens. O resultado desta classificação pode ser usado para simplificar a implementação de mecanismos de recuperação e sua repercussão sobre o uso de recursos do sistema. A conseqüência negativa da implementação de mecanismos de recuperação vinculados à aplicação, é que isso pode causar a diminuição no grau de separação de interesses, exigindo que o projetista da aplicação tenha um conhecimento básico de recuperação e das características que podem interessar a esta. A proposta de Brzezinsky permite antever que a forma de obtenção das informações possa acrescentar complexidade ao desenvolvimento da aplicação, pois a interface terá que classificar as mensagens, mas não desvia a atenção do programador para estes detalhes.

De forma geral, apenas quando estes mecanismos que provêem o suporte à recuperação estão implementados em níveis mais baixos, poderiam passar despercebidos a quem desenvolve aplicações - mas acabam carecendo de eficiência. Assim, um outro enfoque da questão é o de passar informações da aplicação ao nível do sistema operacional, para que ele possa melhorar seus parâmetros de desempenho. A busca de informações da aplicação poderia ser uma forma de suprir estas deficiências, oferecendo parâmetros adicionais para a tomada de decisões pelo sistema operacional; e o ideal seria se isso pudesse ocorrer de forma transparente ao programador.

Huang and Kintala [HUA 93] publicaram um outro artigo onde essas idéias semelhantes são discutidas. Eles demonstraram que tolerância a falhas não é feita de forma integral sem o conhecimento e o auxílio do *software* de aplicação: eles defendem que algoritmos de detecção e de recuperação implementados nos níveis mais baixos não são adequados aos níveis superiores. Esta situação também foi exemplificada com auxílio de um cenário de recuperação, comparando esquemas genéricos, que salvam todos os dados da aplicação residentes na memória, enquanto que os métodos

controlados pela aplicação podem salvar apenas os dados críticos, pois conseguem identificá-los.

Considerando o exposto, realizou-se uma pesquisa para identificar como poderiam ser obtidas informações que pudessem ser exploradas em diferentes níveis do sistema, resultando na identificação de abordagens de captura de mensagens. A principal motivação estava vinculada à obtenção de informações da aplicação para uso em algoritmos de recuperação desenvolvidos paralelamente no Grupo de Tolerância a Falhas da UFRGS.

Os enfoques encontrados na literatura podem ser expressos através de uma das seguintes abordagens: integração [MAF 95a], serviço [FEL 98a] e interceptação [NAR 99]. A abordagem de integração tem como idéia-chave a modificação da camada de comunicação existente, possibilitando o acesso a informações provenientes da aplicação. Outra opção para a obtenção de informações das aplicações independe da camada de comunicação é a abordagem de serviço. O serviço é definido em termos da interface entre a aplicação e a camada básica de comunicação, valendo-se das primitivas de comunicação da camada correspondente. Um exemplo de emprego desta abordagem é o uso com o propósito de viabilizar comunicação de grupo. A abordagem de interceptação utiliza-se de "ganchos" (*hooks*) oferecidos por alguns sistemas operacionais como, por exemplo, a chamada de sistema para embutir mecanismos de interceptação de baixo nível para captura de mensagens.

Estas abordagens são analisadas em publicações, tais como: as de autoria de Felber [FEL 98], Montresor [MON 99a], Narasimhan [NAR 99] e Lung [LUN 2000a]. Nestes trabalhos, são abordadas as características de transparência, facilidade de uso, portabilidade e modularidade. Além dessas são analisadas funcionalidades tais como segurança (e especificamente criptografia) e tolerância a falhas (especificamente o mecanismo de replicação).

Em alguns dos trabalhos citados, os autores apresentaram comparação qualitativa entre as diferentes abordagens, mas não se encontrou uma avaliação numérica relacionada à influência de utilização desses mecanismos sobre o restante do sistema, ou de comparação quantitativa entre os diferentes enfoques. Isso passou a ser objetivo da pesquisa aqui relatada, pois sentiu-se a necessidade de obter mais informações sobre as abordagens no que diz respeito a sua implementação e as conseqüências advindas de sua utilização junto à aplicação.

Esta dissertação tem por meta apresentar a implementação dos protótipos e a análise dos resultados obtidos. Os parâmetros analisados foram divididos em dois grupos: qualitativos e quantitativos. A análise qualitativa tem como ponto de partida os aspectos analisados por Felber [FEL 98a]. Essa análise considera parâmetros tais como: modularidade, transparência e facilidade de uso. O objetivo desta parte do estudo comparativo é o de repensar as conclusões expostas na referência citada, a partir da implementação completa dos mecanismos. Vale lembrar que, no trabalho de Felber, não há informações detalhadas sobre aspectos de implementação.

A análise quantitativa pretende tirar conclusões referentes ao grau de interferência na aplicação. Torna-se importante esclarecer que o objetivo da análise

quantitativa é explorar comparativamente os mecanismos de obtenção de informações contidas nas mensagens trocadas pelas aplicações distribuídas. Não faz parte do escopo do trabalho a classificação dessas informações ou a análise das funcionalidades, que poderiam ser obtidas, como o uso das informações contidas nas mensagens, e identificadas posteriormente ao processo de captura.

Devido aos objetivos comparativos, foi estabelecido o uso do mesmo conjunto de aplicações para avaliação dos diferentes enfoques. Para a execução deste trabalho, então, foi definido o uso da linguagem Java para as aplicações e a implementação dos mecanismos no sistema operacional Linux. Cabe ressaltar que possivelmente, a comparação poderia ser prejudicada se fossem usados conjuntos de ferramentas diferentes e aplicações distintas para o estudo. O sistema operacional foi escolhido como nível de implementação visando atender alguns requisitos importantes para o desenvolvimento do trabalho tais como: a facilidade de instalação, boa documentação, código aberto. Este último foi um fator de essencial para o desenvolvimento do trabalho, pois a implementação realizada neste trabalho exige que o ambiente escolhido possibilite a programação em diferentes níveis: dentro do sistema, e em camadas entre o sistema e a aplicação. Outro fator que reforçou a opção pelo Linux foi a localização de alguns trabalhos que fazem uso de mecanismos similares ao utilizados nesse trabalho, como: Ufo [ALE 99], Janus [WAG 99], SLIC [GHO 98].

Em seguida, implementou-se protótipos das abordagens citadas, através dos quais fosse possível tomar medidas, estudá-las e compará-las. A ênfase, aqui empregada, restringiu-se à comparação entre métodos - e não especificamente à sua influência sobre aplicações em geral. Para estudar este segundo caso, adicionalmente ao que já foi aqui exposto, seria necessário modelar o perfil de diferentes aplicações - no tocante a troca de mensagens - para depois observar esse comportamento combinando de forma exaustiva a alguns tipos de aplicações e abordagens, portanto não haveria tempo para tal.

1.1 Organização do trabalho

Após este capítulo inicial, a presente dissertação está organizada conforme segue.

O capítulo 2 apresenta os aspectos conceituais e qualitativos das abordagens encontradas na literatura. Assim, são reproduzidos os princípios básicos das formas usadas para extrair informações da aplicação, seguidas das respectivas ferramentas que as empregam, usadas como exemplo por Felber em sua classificação. Por fim, é apresentada a análise realizada por Felber, que critica as diferentes abordagens quanto a aspectos qualitativos.

O capítulo 3 trata do estudo de casos, onde são descritos vários projetos identificados através da pesquisa bibliográfica, que utilizam alguma das abordagens para adicionar as mais diferentes funcionalidades.

No capítulo 4, estão descritos alguns recursos estudados e usados para o desenvolvimento dos protótipos: o sistema operacional Linux, características de seu

código fonte, aspectos sobre o *kernel,* conceituação e algumas diferenças entre um módulo e um aplicativo, chamadas de sistema e, de forma mais detalhada, a chamada usada no protótipo de interceptação - a chamada *ptrace*. Finaliza o capítulo a descrição das aplicações utilizadas nos experimentos.

O capítulo 5 descreve o desenvolvimento dos protótipos e a avaliação das abordagens. Assim, são tratados mais especificamente aspectos como a facilidade e a complexidade encontradas na implementação das diferentes abordagens, bem como os resultados obtidos nos experimentos que estão presentes nas análises quantitativas e qualitativas. Portanto, este capítulo contém as informações que compõem um perfil das três abordagens, inicialmente associadas a cada uma, e depois, comparando-as.

Encerram o trabalho as conclusões obtidas através do seu desenvolvimento e sugestões de outros trabalhos que podem dar seguimento a este ou empregando idéias aqui contidas.

O código-fonte dos diferentes protótipos, implementados em Java, figura no anexo.

2 Abordagens utilizadas para captura de informações da aplicação

Nos últimos anos, a computação distribuída tem sido uma das principais tendências da indústria de computadores. A cada dia, mais e mais computadores são ligados às redes exigindo conseqüentemente o desenvolvimento de novas aplicações e serviços distribuídos. Um dos fatos que contribuíram para este interesse crescente foi a enorme expansão da *Internet* [FEL 98a], resultando em um novo desafio para os desenvolvedores de *software.* Visando obter o melhor dos sistemas distribuídos disponíveis, as aplicações começaram a tirar proveito real deste ambiente, tornando-se efetivamente cooperativas e também mais complexas, dificultando sua implementação. Para trabalhar com esta complexidade, explorando todo seu potencial, é exigido um novo paradigma de programação que suporte o desenvolvimento de aplicações distribuídas.

Com o objetivo de concomitantemente atende este novo paradigma e reduzir a complexidade no desenvolvimento de aplicações distribuídas surgiram vários ambientes de programação: esses ambientes são agrupados sob o termo *middleware* [FEL 98a]. Eles se encontram entre os programas da aplicação e os serviços do sistema operacional, fornecendo facilidades em alto nível para o desenvolvimento dessas aplicações, sem ter que se preocupar com detalhes de baixo nível, como comunicação remota e localização de objetos [MON 99b]. Estas plataformas *middleware* oferecem *frameworks* para integração de componentes distribuídos heterogêneos. Dentre esses ambientes estão o CORBA, o DCOM da *Microsoft* e Java RMI.

Eles estão baseados em conceitos de orientação a objetos como: abstração, encapsulamento, herança e polimorfismo, e habilitam interações entre objetos distribuídos-clientes e -servidores. Estas interações tornam-se acessíveis através da utilização de interfaces bem definidas. É permitido aos objetos-cliente terem acesso a serviços providos por objetos servidores através de invocações de métodos remotos [MON 99b, MON 99c].

Porém, estes *frameworks* somente realizam transações com invocações para objetos individuais, enquanto que diferentes tipos de aplicações requerem comunicação um para muitos. Uma extensão interessante para esses *frameworks* consiste em acrescentar primitivas de comunicação de grupo. A idéia-chave da comunicação de grupo é reunir um conjunto de processos ou objetos sob um grupo lógico, e comunicar com todos os membros do grupo ao mesmo tempo com várias ordens garantidas. Diferentes ferramentas de comunicação de grupo são utilizáveis, mas muitas são inadequadas para o desenvolvimento em *frameworks* baseado em objetos. Adicionar comunicação de grupo em um ambiente *middleware* como CORBA permite desenvolver aplicações para beneficiarem-se do poder dos grupos (alta disponibilidade, tolerância a falhas, etc.) enquanto provêem características chaves de um ambiente *middleware* (desenvolvimento simples, transparência na distribuição, integração de componentes, etc.).

Neste contexto, localizaram-se, na literatura, três abordagens básicas: a de **Integração** [FEL 98a, NAR 97a, NAR 97b, NAR 99], a de **Serviço** [FEL 96, FEL 97a,

FEL 97b, FEL 98a, FEL 99] e a de **Interceptação** [FEL 97b, FEL 98a, FEL 99, NAR 97a, NAR 97b, NAR 99]. Vários estudos trazem análise sobre essas abordagens como os apresentados por Montresor [MON 99a], Narasimhan [NAR 97a, NAR 97b, NAR 99], e Felber [FEL 98a, FEL 99]. Essas análises apresentam aspectos qualitativos, pois abordam vantagens e desvantagens em relação a algumas características como transparência, facilidade de uso e modularidade.

Este capítulo tem como objetivo apresentar uma visão geral de um desses estudos, mais especificamente o que contribuiu de forma mais expressiva na realização desse trabalho: a análise realizada por Felber [FEL 98a]. Para tanto, serão reproduzidos os conceitos das abordagens, assim como as respectivas ferramentas estudadas, e por fim a análise que apresenta os aspectos qualitativos das diferentes abordagens.

2.1 Abordagem de integração

A abordagem de integração tem como idéia-chave fazer com que o processamento de grupo seja suportado por um sistema de comunicação de grupo abaixo da camada básica de comunicação. A figura 2.1 ilustra a execução de uma requisição de um grupo de objetos do lado cliente. Esta requisição, quando passada à camada básica de comunicação, é reconhecida como endereçada a um grupo, sendo convertida para uma chamada do tipo *multicast*, pelo conjunto de ferramentas de comunicação de grupo. Logo após, a operação requerida é invocada pela camada básica de comunicação a cada membro do grupo de objetos servidores e por fim, as respostas são coletadas do lado do cliente.

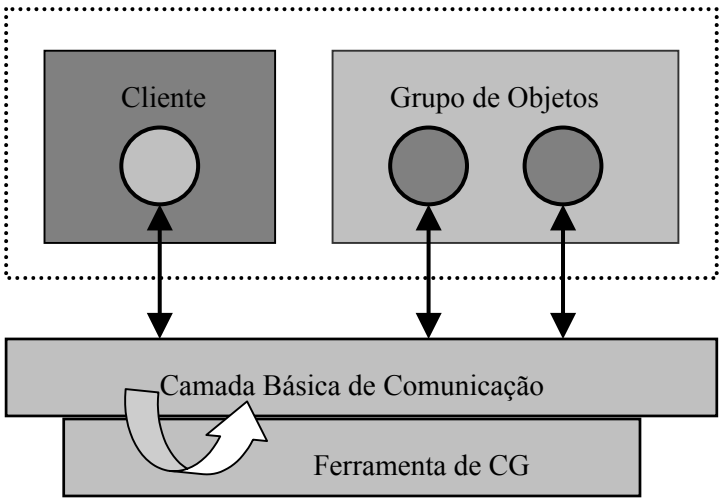


FIGURA 2.1 - Abordagem de integração.

Essa abordagem é caracterizada pela modificação da camada básica de comunicação existente, adicionando um conjunto de ferramentas de comunicação de grupo. Essa modificação, por sua vez, possibilita que uma aplicação seja capaz de fazer uma referência a um objeto sem ter que distinguir entre um objeto simples e um grupo de objetos. Uma referência a um objeto do tipo **t** é mapeada para um grupo de objetos do tipo **t**; isto é, ela é reconhecida pela camada básica de comunicação como sendo uma

referência a um grupo de objetos, oferecendo assim um alto grau de transparência para a aplicação.

Um ponto importante oferecido por esta abordagem e que merece ser destacado é a transparência. A transparência não só oferece a capacidade de referenciar um grupo de objetos como uma única invocação, como se estivesse invocando um objeto simples, mas também é utilizada para manter a invocação de grupo transparente, com uma única resposta retornada ao cliente. Porém, o cliente pode ter acesso a todas as respostas, se for conveniente [FEL 96]. Um outro ponto importante é que a inserção das ferramentas de comunicação de grupo na própria camada básica de comunicação pode tornar a aplicação mais eficiente, pois não é necessário nenhum objeto intermediário para tratar da comunicação como ocorre, por exemplo, na abordagem de serviço, que será estudada na seção 2.2.

Por sua vez, a transparência também tem seus custos ou desvantagens. Um deles é que a semântica de referência de alguns objetos é modificada, tendo como conseqüência a perda da portabilidade, portanto a implementação fortemente dependente da camada básica de comunicação [FEL 96, FEL 97a]. Da mesma forma, a interoperabilidade é prejudicada, pois exige que tanto clientes como servidores usem a mesma implementação para comunicação, dependente de mecanismos de comunicação em grupo específicos a determinados fabricantes.

Existem diferentes variações da abordagem de integração, as quais serão apresentadas no capítulo 3.

2.1.1 Electra

O EOM (*Electra Object Model*) é uma representação abstrata dos mecanismos básicos necessários para a construção de aplicações em ambientes distribuídos assíncronos, constituídos de objetos que são executados em uma coleção de máquinas, as quais podem interagir através de troca de mensagens ou por meio de procedimentos de chamada remota [MAF 95a, MAF 95b]. Além disso, o Electra objetiva suportar programação distribuída orientada a objetos, permitindo aos programadores alcançar alta qualidade nas aplicações e a reutilização de componentes de *software*. O Electra é baseado no modelo de grupo de objetos, sendo implementado como uma extensão do CORBA *Basic Object Adapter* (BOA) [MAF 96]. Esta extensão ou modificação é realizada com a integração de mecanismos de comunicação de grupo ao ORB. Essa extensão do ORB é a idéia básica da abordagem de integração.

O Electra pode ser executado tendo como suporte a vários ambientes de comunicação de grupo tais como Horus, Isis e MUST. Segundo Maffeis [MAF 95b, LAN 97], o Electra pode ser portado facilmente para Amoeba, Chorus, Transis e outras plataformas que provêem *multicast* e *threads*. Embora o Electra possa ser configurado para executar diretamente sobre sistemas operacionais como UNIX ou Windows NT, conforme Maffeis [MAF 95a], é preferível uma solução onde seja empregada uma plataforma como Horus ou Isis, visto que o sistema tem habilidade de explorar o suporte oferecido de comunicação de grupo confiável e sincronismo virtual como uma parte fundamental do ORB.

O Electra tem a sua arquitetura formada por camadas conforme demonstrado na figura 2.2. Estas camadas são as seguintes: CORBA *Static Invocation Interface* (SII), *Object Request Broker Interface* (ORB) e a *Basic Object Adapter* (BOA), as quais são baseadas na *Dynamic Invocation Interface* (DII) que pode ser vista como o núcleo do ORB. Estes componentes são equipados com operações para dar suporte à comunicação de grupo. O núcleo é construído sobre um módulo de *multicast* RPC, que executa RPC assíncrono para objetos simples e para grupos. Uma abordagem direta consiste em construir um módulo de *multicast* RPC sobre Horus, porém, devido ao propósito de dispor flexibilidade e portabilidade, a arquitetura do módulo-base é formada por camadas independentes do ambiente, chamadas de máquina virtual.

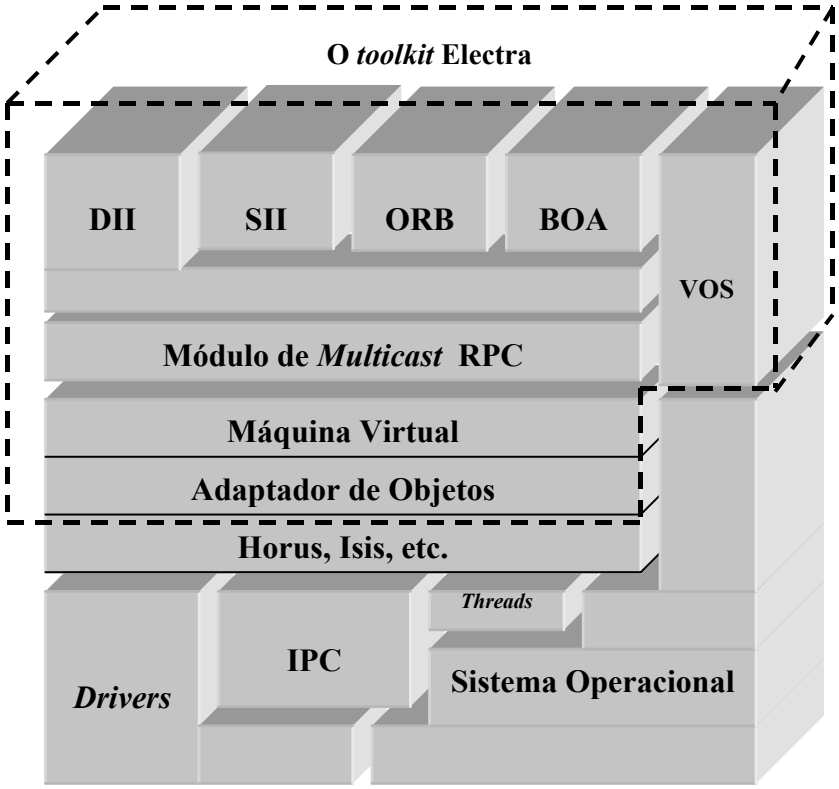


FIGURA 2.2 - Arquitetura do Electra.

A interface da máquina virtual disponibiliza operações para criar *endpoints* de comunicação, para agregar *endpoints* para grupos, para enviar mensagens assíncronas e para processos leves. O módulo de *multicast* RPC e a máquina virtual comunicam-se por meio de *downcalls* e *upcalls*. O *Virtual Operating System* (VOS) é uma camada virtual do sistema operacional usada pelas as aplicações para interagirem com o sistema operacional propriamente dito, assegurando sua portabilidade e um tratamento *thread-safe* [MAF 95b]

Para mapear a interface da máquina virtual sobre o API específico a uma determinada ferramenta depende da implementação de um adaptador de objetos. Um adaptador de objetos encapsula todo o código do programa que é específico a uma ferramenta e necessário como suporte ao módulo de *multicast* RPC. Para portar o Electra para uma nova ferramenta, os programadores precisam apenas desenvolver ou modificar apropriadamente um adaptador de objetos, como os que existem para o Isis e

para o Horus. Isto faz com que não seja necessária a re-compilação das aplicações e torna o Electra portável a outras ferramentas.

O ORB é o coração do modulo de comunicação. Provê uma infra-estrutura que permite aos objetos comunicarem-se de forma independente de uma linguagem de programação específica e da técnica usada para implementar os objetos, isto é, o ORB é portável [MAF 95b]. Os objetos não têm um papel fixo: em um momento podem representar um cliente e, em outro momento, um servidor. Objetos comunicam-se no Electra através de invocação de *multicast* confiável ou ponto a ponto. Os programadores podem referenciar grupos de objetos ou objetos simples usando a mesma interface. A invocação *multicast* pode ser emitida por ambas as interfaces de invocação CORBA, estática ou dinâmica. A comunicação dos grupos de objetos pode ser executada através de duas formas:

- na forma transparente, o grupo de objetos aparece como sendo um objeto simples altamente disponível;

- na forma não transparente, permite ao programador ter acesso ao resultado de uma invocação que foi produzida pelos membros do grupo, individualmente.

No Electra, a interface de invocação estática é baseada na interface de invocação dinâmica; por exemplo, o *stub* aloca um objeto do tipo *Request* para executar invocações. O Electra não requer qualquer modificação na DII padrão [MAF 95a]. Os principais componentes de uma interface de invocação dinâmica de uma classe que representa uma requisição a ser enviada a um objeto CORBA [MAF 95a] são ilustrados a seguir:

1. são utilizados os parâmetros do construtor para especificar o destino da invocação do objeto, a saber:
   - a operação a ser invocada (identificador);
   - um objeto de ambiente para garantir uma execução resultante da operação;
   - o contexto da operação CORBA;
   - e uma região de memória para armazenar os resultados da operação.
2. a operação de adição de argumento é executada para ordenar os argumentos de cada invocação;
3. *send* executa uma invocação assíncrona para determinar se uma invocação remota foi completada, apura as respostas que devem ser invocadas e suspende as chamadas até que a operação seja completada.

No Electra, a operação *send_oneway* executa o mesmo tipo de invocação remota que *send*. Em contraste à especificação CORBA, no Electra a operação é confiável durante a execução de uma invocação síncrona, a chamada fica suspensa até o término da operação. Quanto ao ORB padrão, não é exigida qualquer modificação para dar suporte ao modelo de objetos do Electra.

O módulo de *multicast* RPC é um módulo de RPC genérico, situado no coração da arquitetura do Electra [MAF 95a]. É ele quem conduz a entrega de baixo nível das

invocações remotas a objetos. A sua principal função é tornar possível o uso de RPC assíncrono para ambos os destinos: objetos simples ou grupos. Os mecanismos de invocação síncronos são providos pela DII, que reside no topo do módulo de RPC. A diferença entre uma RPC e uma invocação remota a objetos é que, enquanto o modelo de comunicação da RPC expressa uma abstração de baixo nível, a invocação remota a objetos tem uma assinatura que obedece a declaração de interface relacionada, além de efetuar verificação de tipos de argumentos de operação, dando a ilusão de que a comunicação se efetua entre objetos.

O módulo RPC trabalha com mudanças na composição dos grupos e com o controle de respostas de uma forma independente do subsistema de comunicação empregado. É baseado somente na interface da máquina virtual (VM). A VM usa *upcalls* para informar ao módulo RPC a chegada de mensagens, a troca de visão do grupo e ocorrência de defeitos; e usa *downcalls* para alocar *endpoints* de comunicação, registrar as *threads* no destinatário e enviar mensagens. Na versão atual do Electra, só há uma instância do módulo RPC por processo. Com isto, todos os objetos em um processo interagem com o mesmo módulo de RPC.

No Electra, a interface do adaptador BOA contém operações especiais para trabalhar com comunicação de grupo. Foi adicionada a capacidade de criar, destruir, unir-se e sair de um grupo. Para criar objetos, a operação de criar é emitida com parâmetros de referência a objeto. A referência pode ser instalada em um servidor de nomes ou convertida em uma representação literal (*string).* A política de argumento é utilizada para informar ao ambiente que dá suporte à comunicação de grupo o tipo de protocolo de *multicast* que será empregado para ordenamento total ou causal [LAN 97, MAF 95b]. Quando o Electra está configurado para executar no Isis, a política de objetos seleciona protocolos *abcast, cbcast*, ou *gbcast* para executar *multicast*. Na configuração para o Horus, a política de objetos serve para conduzir a um dos protocolos suportados pelo Horus, além de apoiar vários protocolos de ordenamento e protocolos confiáveis, como também as comunicações em cima de UDP, IP-*multicast,* ATM e outros. No Electra, para inserir uma implementação de objetos em um grupo é usada uma operação *join*; para remover um objeto do grupo usa-se uma operação *leave.* Para a extinção de um grupo existe a operação *destroy_group*.

Objetos, que se unem a um grupo freqüentemente, devem ser iniciados com um estado dependente da aplicação. Em replicação ativa, por exemplo, todos os membros de um grupo de réplicas precisam ter o estado interno sincronizado e os recém chegados precisam obter o estado atual antes de participarem de uma computação redundante. No Electra, a transferência de estado é realizada com a ajuda do BOA [MAF 95b], através dos métodos *get_state* e *set_state.* Quando um objeto une-se ao grupo, a máquina virtual invoca um método *get_state* de um membro qualquer do grupo para obter seu estado interno. Em colaboração ao protocolo de *membership* do grupo, esta informação é transmitida aos recém-chegados e a operação *set_state* é invocada. Além disso, enquanto ocorre uma troca de visão, a operação *view_change*, em cada membro é invocada automaticamente. A visão do objeto contém informações sobre o número real de membros e qual objeto está sendo removido ou juntou-se ao grupo.

2.2 Abordagem de serviço

Uma metodologia alternativa para prover comunicação de grupo é através da abordagem de serviço, a qual tem como idéia básica à capacidade de oferecer comunicação de grupo como um serviço acima da camada básica de comunicação e não como parte da própria, conforme descrito em diversos documentos produzidos por Felber *et al.* [FEL 98a, FEL 97a, FEL 97b]. Com essa abordagem, acrescentam-se novas funcionalidades além da comunicação de grupo, como serviços de transação e o serviço de persistência [FEL 98a].

A representação apresentada na figura 2.3 mostra grupos e um conjunto de objetos de serviços, acima da camada básica de comunicação; esses últimos são responsáveis pela gestão de grupos e ainda pela entrega de mensagens aos membros do grupo de objetos. Assim sendo, as requisições dos objetos clientes passam através dos objetos de serviço que fazem a difusão das mensagens.

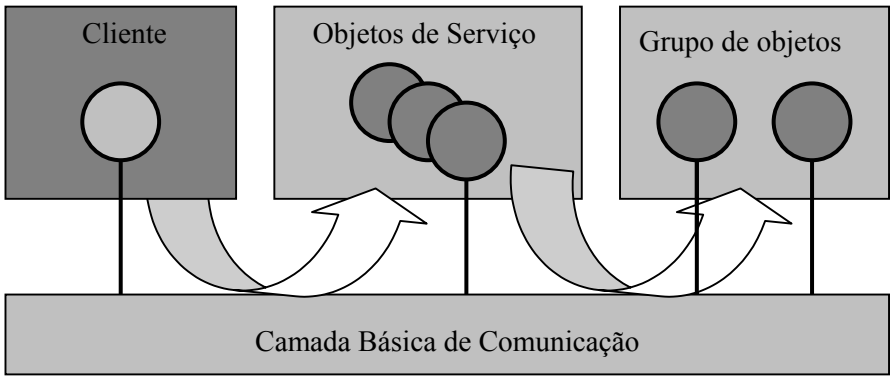


FIGURA 2.3 - Abordagem de serviço.

Com esta metodologia, o serviço não é dependente da camada básica de comunicação. O serviço é definido em termos de uma interface e, por utilizar as primitivas de comunicação da camada básica de comunicação, beneficia-se da propriedade de interoperabilidade.

Na abordagem de serviço, um fato relevante para tolerância a falhas é que o serviço não necessita ser centralizado. Ele pode ser composto por vários objetos, localizados em diferentes nodos da rede. Esses objetos trabalham juntos para prover um serviço completo [FEL 98a], e assim também evitando ter um ponto único de falha. Na abordagem de serviço, com a intenção de obter um nível de transparência semelhante à da abordagem de integração, foi necessária a utilização de objetos *proxies*[1] para fornecer às aplicações uma interface para o serviço de grupo.

[1] Um *proxy* é uma entidade que atua como um representante local para serviços remotos. Neste sentido, para o cliente um *proxy* comporta-se como se a implementação do serviço que ele representa estivesse no próprio serviço do cliente.

2.2.1 OGS

O OGS (*Object Group Service)* é um serviço que gerencia grupos de objetos CORBA e provê primitivas para comunicação com esses grupos em um ambiente CORBA [FEL 98a]. O OGS é composto de um conjunto de interfaces IDL (*Interface Definition Languages*) genéricas que fornecem suporte a grupos de objetos provendo facilidades para a computação distribuída confiável. O OGS permite aos clientes enviarem invocações para grupos de objetos sem saber o número e identidade dos membros do grupo [FEL 98b, FEL 98c]. Ele também provê apoio para invocação a grupos de forma transparente, permitindo aos clientes invocar operações em grupos de objetos como se eles estivessem invocando um simples objeto. OGS é baseado apenas em mecanismos do padrão CORBA, sendo assim, portátil a qualquer implementação de ORB. O OGS pode ser usado por qualquer linguagem de programação que é apoiada por CORBA, ou por qualquer sistema que apóia o CORBA *Internet Inter-Orb Protocol* (IIOP), segundo descrito por Felber [FEL 98a].

O ambiente do OGS oferece serviços para objetos que provêem várias facilidades para a computação distribuída confiável tais como: protocolos de consenso e detectores de defeitos de componentes distribuídos, que são usados na implementação de primitivas de comunicação de grupo.

No OGS é utilizada a abordagem orientada a componente, sendo que um componente é uma unidade de trabalho e distribuição e geralmente designa um objeto ou um grupo de objetos colaboradores que possam ser acessados por clientes através de uma interface bem definida. Um componente em CORBA é um bloco de construção que emprega mecanismos orientados a objetos como herança e polimorfismo, para prover serviços específicos [FEL 98a]. A tecnologia de objetos distribuídos permite reunir sistemas de informações cliente/servidor simplesmente juntando ou separando componentes, que possam se tornar uma coleção de componentes colaboradores, os quais permitem modificações individualmente, sem afetar a estrutura dos outros objetos do sistema ou do nodo com os quais interagem.

Tipicamente os sistemas de comunicação de grupo são organizados em uma arquitetura de camadas como *multicast* de grupo*, membership* do grupo, protocolos de consenso, detectores de defeitos e vários outros componentes. O OGS mapeia as camadas da arquitetura de comunicação de grupo usando uma abordagem orientada a componentes, onde os serviços não são apresentados em uma arquitetura de camadas [FEL 98a], mas como um grupo de componentes dispostos ortogonalmente com relação de uso entre eles. Na figura 2.4, é apresentada uma visão abstrata dos principais componentes que definem a arquitetura do OGS. Embora isto não fique claramente visível na figura, cada componente é especificado independentemente e todos eles interagem entre si pelo ORB, sendo que a aplicação pode utilizar qualquer um destes componentes diretamente.

Conceituando consenso, pode-se dizer que é uma forma de proporcionar que vários elementos de processamento alcancem uma decisão comum, de acordo com os valores iniciais, apesar da possibilidade de colapso eventual de alguns deles [FEL 98b]. O problema do consenso é uma abstração central para resolver vários problemas de acordo como o acordo atômico, ordenamento total, *membership* e assim alcançar

tolerância a falhas em sistemas distribuídos. Esses problemas de acordo estão presentes em muitos sistemas baseados em sincronismo virtual e transações ou replicação, e são implementados usando protocolos. Felber [FEL 98a] propôs um serviço de consenso genérico que pode ser usado por vários protocolos. Além da modularidade, esta aproximação habilita a implementação eficiente dos protocolos, como também a caracterização precisa da disponibilidade da aplicação. O consenso é provido no nível de objetos e não no nível de processos, sendo completamente definido em condições de interface IDL e é utilizável entre objetos heterogêneos.

O consenso no OGS é composto basicamente por dois tipos de objetos: os gerentes de consenso e os participantes do consenso [FEL 98a]. Os gerentes são objetos de algum serviço que implementam o protocolo de consenso e alcançam o consenso entre si. Eles agem como uma "caixa preta", e sua implementação é provida por serviço. O estado do protocolo de consenso não é exposto à aplicação, e pode ser mudado sem impacto aos clientes do serviço; os participantes são objetos específicos da aplicação que são envolvidos somente no consenso por propor um valor inicial e receber uma decisão.

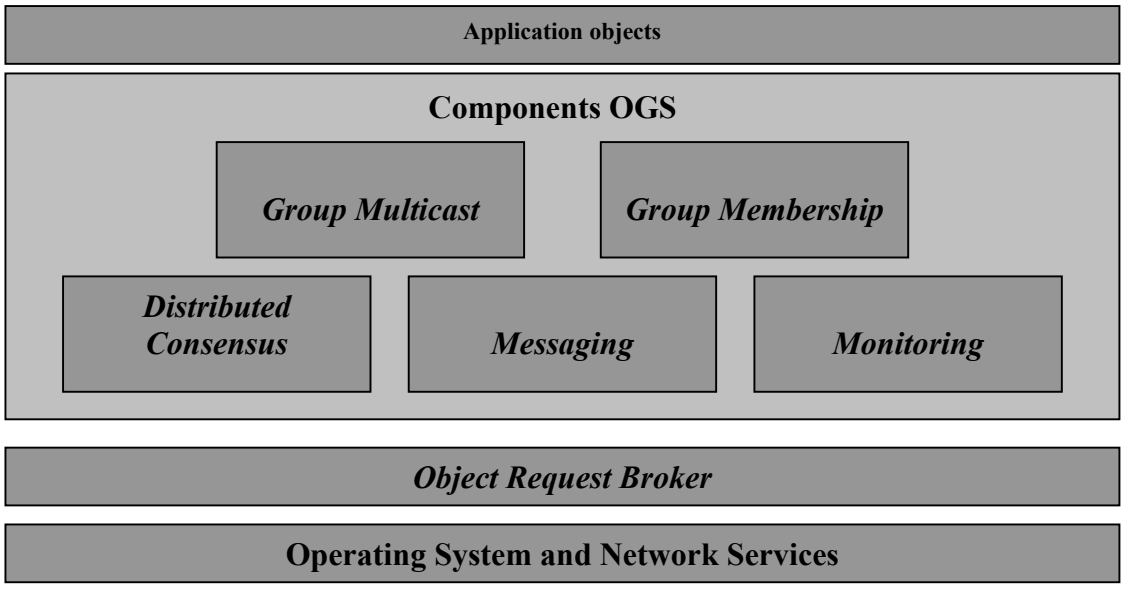


FIGURA 2.4 - Visão geral da estrutura do OGS.

No início de um consenso, os participantes têm que entregar os seus valores iniciais aos gerentes. Isso pode ser realizado de duas formas:

- o participante fornece o seu valor inicial como um parâmetro para a operação que inicia o consenso. Esta primeira alternativa tem a vantagem de adequar-se exatamente ao modelo de Chandra e Toueg, algoritmo de consenso usado no OGS. Este modelo requer que todo participante invoque explicitamente essa operação no gerente de consenso para executar o protocolo, chamado de instalação de consenso explícito.

- a outra forma ocorre nas chamadas do gerente de consenso ao participante para obter o valor. Esta segunda alternativa consiste em ter um gerente de consenso assíncrono, o qual obtém o valor inicial dos participantes depois de ter obtido o consenso. É um modelo mais geral já que todos os

> participantes não precisam iniciar o consenso explicitamente para o protocolo ser iniciado: os objetos participantes podem participar passivamente do protocolo de consenso, chamado de instalação de consenso implícito. Ele exige que os participantes sejam previamente conectados a um gerente de consenso antes da instalação do consenso.

No OGS foi adotada a segunda alternativa devido ao fato, de que possibilita a instalação de ambos os consensos, explícito e implícito, e assim provê mais flexibilidade de implementação.

Conforme citado anteriormente, a implementação do serviço de consenso do OGS está baseada no algoritmo de Chandra e Toueg. Este algoritmo disponibiliza uma solução para os problemas de consenso em sistemas assíncronos com o uso de um detector de defeitos não confiável, conhecido como ✧S, sob a condição de que a maioria dos processos não falha durante a tomada de decisão. O algoritmo encontra-se baseado no modelo do coordenador circular, onde em cada rodada um processo diferente assume o papel de coordenador. Uma rodada consiste em quatro fases, conforme descrito a seguir [FEL 98a]:

1. na primeira rodada todos os processos enviam suas estimativas para o coordenador atual. Nesta rodada, a estimativa é o valor inicial do processo;

2. o coordenador da rodada corrente aguarda pela maioria das estimativas e seleciona a que tem o maior *timestamp*, que representa a última rodada em que o processo mudou sua estimativa. Esta estimativa é enviada para todos os processos;

3. todos os processos aguardam pela nova estimativa proposta pelo coordenador atual. Quando o processo recebe a nova estimativa, responde ao coordenador com uma mensagem de reconhecimento ACK. Caso isto não ocorra, o coordenador é suspeito de falha pelo detector de defeitos e uma mensagem de reconhecimento negativo NACK é retornada ao coordenador;

4. na última fase de uma rodada, o coordenador aguarda pelo reconhecimento da maioria dos processos. Caso nenhuma das mensagens de reconhecimento seja um NACK, o coordenador decide sua estimativa atual e realiza a difusão, de forma confiável, a todos os processos.

A implementação do serviço de consenso no OGS é totalmente orientada a eventos, que é implementado usando a abordagem de máquina de estados e reage a dois tipos de eventos: recepção de mensagens e notificação de suspeitos. Os diferentes estados correspondem às diferentes fases do algoritmo. A implementação permite múltiplas execuções em paralelo do consenso. O consenso é decidido em poucas rodadas. Podem ser executadas instâncias de consenso em paralelo ou seqüencialmente com grupos de participantes completamente separados.

O OGS usa o serviço de consenso para a implementação de *multicast* e visão de *membership*, sendo que o seu objeto é obter informações idênticos no respectivo ordenamento dos eventos, mensagens e trocas de visão, recebidos pelos membros do

grupo [FEL 98a]. Para garantir as exigências destes algoritmos, o algoritmo de Chandra e Toueg foi expandido de tal forma que possa, opcionalmente, decidir em uma composição de grupo não vazio, uma maioria de estimativas iniciais, em vez de uma única estimativa. Para que esta nova propriedade fosse possível, a segunda fase do algoritmo original foi modificada.

A modificação ocorrida na segunda fase do algoritmo de consenso foi a seguinte: nesta fase, o coordenador propõe uma lista de todas as estimativas recebidas dos outros participantes, se todas as estimativas tem um *timestamp* zero. Caso contrário, é selecionada a estimativa com *timestamp* mais alto, como ocorre no algoritmo original. A regra de validade do consenso deve ser alterada. Na regra original, se todos os participantes que propõe um valor, propõem **v**, então todos os processos corretos optarão por **v** em um tempo finito. Na nova regra, se um processo decide **v**, então **v** é uma lista não vazia que contém a maioria dos valores iniciais, dos quais cada um foi proposto por algum processo.

O algoritmo de Chandra e Toueg é utilizado pelo OGS para *multicast* confiável [FEL 98a]. Este algoritmo usa o reenvio de mensagens para assegurar que todos os processos destino corretos recebam a mensagem. A idéia desse algoritmo é a seguinte: o processo, ao receber uma mensagem de *multicast* confiável pela primeira vez, retransmite a mensagem para todos os outros processos destino e então entrega à aplicação. Embora não seja muito eficiente este mecanismo assegura que todos os processos corretos entreguem todas as mensagens. O algoritmo é definido em torno de duas primitivas: R-*multicast* e R-*deliver*.

No algoritmo de ordenação total e visão de *membership,* cada participante do grupo de mensagens não ordenadas é atualizado a cada instante que uma mensagem é recebida e a lista dos membros suspeitos da visão atual é atualizada a cada momento. O detector de defeito notifica os participantes sobre um suspeito ou sobre um não suspeito. É usada uma variável como identificador do consenso para sincronizar as instâncias de consenso executadas por todos os participantes [FEL 98a]. Um consenso é lançado quando há mensagens para ordenar ou há membros para remover da visão atual. Cada processo entrega o grupo de mensagens contidas na decisão numa ordem pré-definida que foi acordada por todos os participantes. Caso o grupo de membros suspeitos participe na decisão, todos os membros corretos instalam uma nova visão sem os membros suspeitos. O processo de unir-se ou deixar um grupo existente difere um pouco da remoção de um membro defeituoso. É implementado por um pedido totalmente ordenado emitido pelo membro que se uniu ou deixou o grupo. Esse pedido é processado por todos os membros do grupo que atualizam suas visões adequadamente. Os membros que se unem ao grupo recebem informações de serviços específicos como parte do protocolo de transferência de estado.

A comunicação de grupo é bem adaptada para a replicação de objetos, porém, não são todos os modelos de replicação que podem ser usados de forma transparente com as interfaces de grupo. Na replicação ativa, que é onde todos os membros têm o mesmo comportamento, o qual é aceitar uma requisição, encaminhá-la e opcionalmente retornar uma resposta, não é necessária a re-implementação de suas operações. Para códigos que implementam operações específicas de aplicações, a replicação ocorre de forma transparente [FEL 98a]: o OGS invoca diretamente a operação designada e o

pedido de processamento é executado do mesmo modo, independente do objeto ser replicado ou não, sendo que não é necessário nenhum apoio extra do serviço de grupo para a implementação dos servidores de replicação ativa.

No modelo de replicação primário-*backup,* que também é chamado de replicação passiva [FEL 98a, DEF 98], existe uma distinção entre o objeto primário que processa as requisições e as réplicas, as quais só recebem a atualização do primário. O mecanismo de atualização envia ao cliente as modificações que o processamento induziu no estado do primário, enquanto as réplicas podem conter dados completamente desatualizados. Esta diferença de papéis requer uma interface de suporte para ser adicionado ao serviço de grupo. Existem várias formas de fazer isto: o OGS optou pelo limite do uso da replicação passiva para aplicações não tipadas e adaptar a interface do OGS para trabalhar com transmissão de atualização do primário para as réplicas.

O OGS possui interfaces de subclasses do serviço de grupo para trabalhar com replicação primário-*backup*, que define uma nova semântica de comunicação. Para trabalhar com a transmissão de atualização é definida uma estrutura que contém tanto a resposta para o cliente como a informação de atualização para o *backup.* Essa estrutura é resultante da operação de entrega que é invocada na cópia primária, e o *backup* recebe a informação de alteração de estado através da operação de atualização. O OGS, através de suas interfaces, torna possível a utilização dos dois modelos de replicação alternativamente tempo devido ao serviço de grupo suportar ambos os modelos.

2.3 Abordagem de interceptação

Na abordagem de interceptação, representada graficamente na figura 2.5, as mensagens enviadas aos objetos servidores devem ser capturadas através de um mecanismo de baixo nível e mapeadas para um sistema de comunicação de grupo como, por exemplo, realizado no sistema Totem [FEL 98a, NAR 99]. Esse mecanismo encontra-se separado da camada básica de comunicação e a captura da mensagem é realizada de forma transparente para a aplicação.

Alguns sistemas operacionais provêem "ganchos" (*hooks*) que possam ser explorados para desenvolver componentes com o conceito de interceptadores (*interceptors).* O sistema operacional UNIX apresenta duas implementações possíveis [NAR 99]; a chamada de sistema, que provê a interceptação ao nível do sistema, e a biblioteca de rotinas, que provê a interceptação em nível de rotinas de bibliotecas.

Quando uma aplicação realiza uma solicitação de serviço do sistema operacional, a execução ocorre através de **chamada de sistema**. Uma chamada de sistema corresponde a um procedimento de uma biblioteca de rotinas que a aplicação pode chamar. Esta **biblioteca de rotinas** cria uma abstração para a instrução de interrupção*,* (chamada protegida a procedimentos) fazendo-a parecer uma chamada a procedimento comum [TAN 92].

Os chamados "ganchos" disponibilizados por alguns sistemas operacionais para a interceptação de mensagens, que ocorre quando elas estão em cima da IIOP, causam o desvio das requisições antes de chegar à camada TCP/IP. É uma solução dependente das

funcionalidades oferecidas pelo sistema operacional; portanto apresenta um problema de interoperabilidade [NAR 97a, NAR 97b].

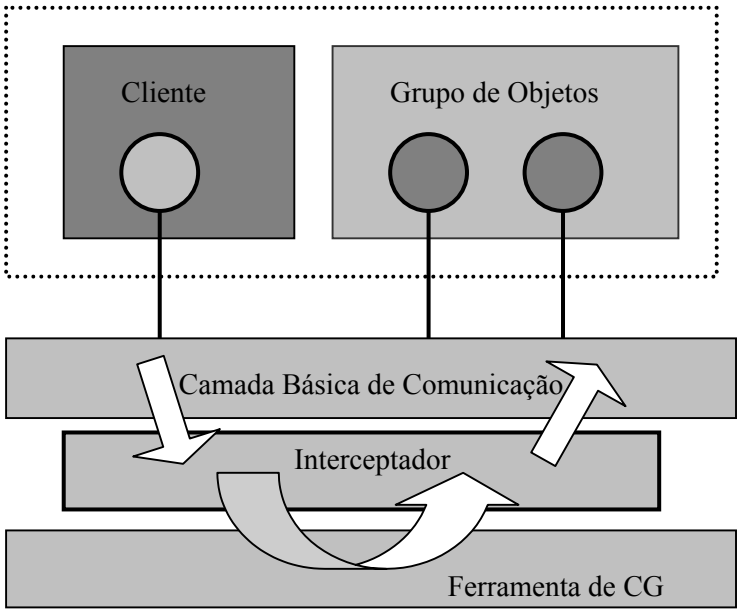


FIGURA 2.5 - Abordagem de interceptação.

Um interceptador, por exemplo, pode ser usado para adicionar um componente de compressão, que transparentemente modifica as mensagens que são trocadas entre os objetos de uma aplicação CORBA, enquanto o comportamento da aplicação permanece inalterado [NAR 97a, NAR 99]. A maioria dos dados são trocados entre objetos da aplicação por mensagens IIOP. Assim, a interface do interceptador é direcionada para a comunicação de mensagens IIOP.

O processo de compressão ocorre da seguinte forma:

- do lado do emissor:
  1. o interceptador captura a mensagem IIOP enviada pelo emissor, e passa para os componentes de compressão;
  2. o componente de compressão realiza a compressão dos dados e retorna para o interceptador;
  3. o interceptador passa a mensagem IIOP, com os dados compactados, para o ORB;
  4. o ORB envia a mensagem via TCP/IP.

- do lado do receptor:
  1. interceptador captura a mensagem IIOP que recebeu via TCP/IP, e passa a mensagem para um componente de descompressão;
  2. o componente de descompressão descompacta a mensagem produzindo a mensagem IIOP original;
  3. o interceptador devolve a mensagem do IIOP ao ORB;
  4. o ORB entrega a mensagem ao objeto designado.

Os objetos (emissor e receptor) ignoram a presença do interceptador e dos componentes de compressão e descompressão. Além disso, para o ORB é transparente a presença dos componentes de compressão e descompressão e ele trabalha com as mensagens de IIOP que contêm os dados compactados da mesma forma que trabalha com as mensagens de IIOP sem dados compactados [NAR 99].

Um interceptador também pode ser utilizado para viabilizar segurança para a aplicação através de componentes que codificam e decifram, autenticam e verificam ou rejeitam conexões com base nas especificações do usuário ou na configuração da política de segurança.

Os componentes de codificação e decodificação modificam as mensagens IIOP trocadas pelos objetos da aplicação sem haver a necessidade de modificar o comportamento da aplicação. Outros componentes de segurança, como os para autenticação, podem resultar em modificação do comportamento da aplicação.

Os componentes de segurança que codificam e decodificam mensagens da IIOP com a utilização de interceptadores fazem isto da seguinte forma: o interceptador emissor captura a mensagem da IIOP e a passa para um componente que codifica os dados contidos nela. O interceptor envia a mensagem da IIOP que contém os dados codificados para o ORB que repassa via TCP/IP para o objeto designado [NAR 97b]. O interceptador receptor extrai a mensagem da IIOP que contém os dados codificados, obtém a mensagem original que contém os dados decodificados usando um componente decodificador e passa a mensagem para o ORB que entrega ao objeto destino.

Outros componentes de segurança que podem ser usados são os componentes de adaptação protocolares para envio de mensagens da IIOP que usam um protocolo seguro em vez de TCP/IP. Componentes de segurança podem também ser componentes de *profiling* para monitorar a aplicação, ou seja, são componentes que possibilitam construir um perfil de uma determinada execução ou de objetos, sem modificá-los.

2.3.1 Eternal

O *Eternal System* é um *middleware* que trabalha entre o ORB do CORBA padrão e sobre um sistema operacional normal. Ele expande o CORBA sem a necessidade de alterar as aplicações utilizando interceptadores para encadeamento de protocolos, monitoramento, escalonamento e componentes de administração de réplicas e segurança.

Atualmente, os objetos CORBA não interagem com outros tipos de objetos distribuídos sob um protocolo que não se adequar com o GIOP [NAR 99]. Para que isto ocorra, os programadores têm que gastar esforços para reescreverem o ORB a fim de obterem soluções. Uma forma eficiente é desviar as mensagens do GIOP que são enviadas por objetos CORBA, transparentemente e sem modificação do ORB, com componentes que mapeiam o GIOP para outros protocolos e deste modo pode-se construir adaptadores protocolares que tiram proveito de capacidades do CORBA sem modificar a aplicação ou o ORB.

O CORBA provê suporte para muitos dos modelos de *multithreading* comumente usados. Porém, para algumas aplicações é desejável empregar uma política não suportada pelo ORB. Para que isso ocorra de um modo mais fácil, é interessante o uso de um interceptador para introduzir componentes de escalonamento antes da requisição alcançar os objetos destino. Para controlar a ordem na qual as *threads* e as requisições são liberadas pela aplicação, o componente de escalonamento pode impor ao ORB ou a aplicação políticas de escalonamento e execução de *threads*.

Os mecanismos de tolerância a falhas que o CORBA dispõem atualmente são rudimentares e consistem principalmente de exceções retornadas no caso de falha de um objeto ou processador. A utilização de interceptadores adiciona componentes que tratam de replicação de objetos, detectores de defeitos e recuperação de falhas de uma forma transparente à aplicação e ao ORB. Esses componentes, uma vez configurados, operam independentemente do programador da aplicação e do ORB.

O Eternal provê replicação de objetos, mantém a consistência das réplicas de objetos e possibilita a replicação tanto de objetos clientes quanto de servidores [MOS 98]. Os objetos no sistema Eternal podem agir como clientes e como servidores. As réplicas são associadas em grupos; o acesso a estes pode ocorrer através do nome do grupo. O grau, o tipo de replicação e a localização das réplicas são transparentes através da abstração de grupo. O programador não precisa se preocupar com dificuldades da replicação como consistência, distribuição, detecção de defeito e recuperação que são inerentes a essas plicações. O Eternal provê tolerância a falhas e alta disponibilidade para aplicações distribuídas através da replicação de objetos.

Na estrutura do sistema, que é apresentada na figura 2.6, pode-se ver que o Eternal trabalha com implementações de CORBA padrão em cima do sistema operacional UNIX. O sistema Eternal aproveita-se da interface `/proc` do UNIX para monitorar as chamadas ao sistema realizadas por um objeto para estabelecer uma conexão IIOP através do TCP/IP [NAR 97a, NAR 97b], e para enviar mensagens sobre esta conexão. O Eternal intercepta as mensagens de IIOP, antes que elas cheguem ao TCP/IP e as repassa para o sistema de comunicação de grupo Totem [MOS 98, NAR 99], o qual envia as mensagens para os grupos de objetos que contêm as réplicas. Qualquer sistema de comunicação de grupo que apresente garantias semelhantes ao sistema Totem pode ser usado com o Eternal. O Eternal é composto de diferentes componentes como:

- *interceptor,* como o seu próprio nome sugere, é responsável por interceptar as requisições do CORBA padrão;

- *replication Manager* trata as cópias de objetos replicados;

- *resource Manager* administra o sistema com operações como controlar o tipo e número de réplicas;

- *evolution Manager* executa as versões atualizadas automaticamente e a evolução dos objetos em tempo real.

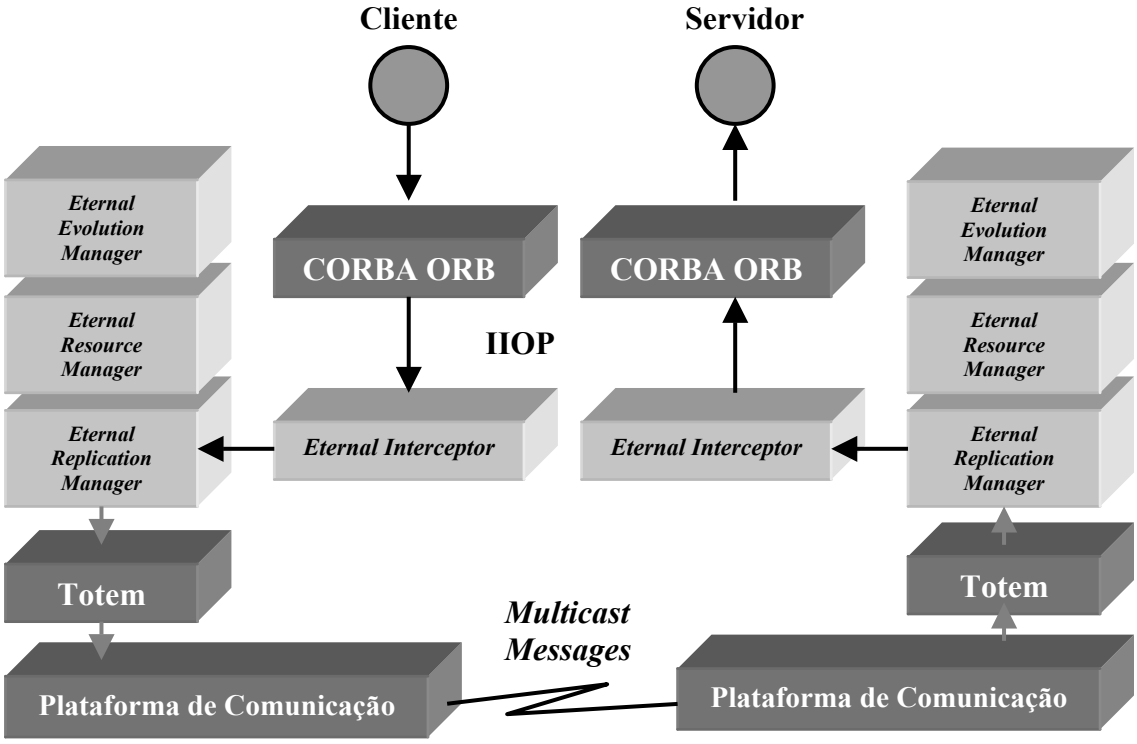


FIGURA 2.6 - Arquitetura do sistema Eternal.

Em sistema como Solaris [NAR 99], o sistema de arquivo `/proc` provê acesso à parte interna de cada processo que está sendo executado na máquina. No diretório `/proc` existe a chamada imagem dos processos; é um arquivo que corresponde à identificação dos processos UNIX. Através dela, pode-se manipular o processo utilizando uma interface padrão que permita capturar de chamadas a sistema de cada processo. Os argumentos e os valores retornados das chamadas de sistema interceptadas podem ser extraídos e modificados e os processos interceptados podem ter seu comportamento alterado. Assim, cada objeto CORBA pode ser monitorado durante sua vida para chamadas de sistema relacionadas a muitas atividades diferentes, inclusive administração de memória, comunicação de rede e acesso a arquivos.

A implementação de interposicionamento de biblioteca explora as facilidades ligação em tempo real do sistema UNIX, que permite adicionar objetos compartilhados a um processo. Esta forma de ligação não requer que a aplicação seja modificada, religada ou re-compilada. O interposicionamento de biblioteca explora o fato de que um executável pode ter símbolos (variáveis indefinidas e funções) e que as definições não são resolvidas até o tempo de execução. Em tempo de execução, o primeiro objeto compartilhado que identifica um símbolo, é aceito como a sua definição. Se os objetos subseqüentes que compartilham o mesmo espaço de processamento também provêem definições para o mesmo símbolo, estas definições são ignoradas pelo ligador em tempo de execução (*runtime linker*). A primeira definição é dita "interposta" a todas as outras definições do mesmo símbolo. Desta forma, um processo pode usar definições de funções providas por uma biblioteca interposta em lugar das definições de funções originais ou padrão. As *DLLs* do *Windows NT* da *Microsoft* provêem "ganchos" semelhantes que podem ser explorados para construir interceptadores [NAR 99].

O sistema Eternal usa técnicas-padrão de replicação (replicação passiva e ativa) para proteger-se de falhas de omissão, que ocorrem quando um objeto ou processador não envia uma mensagem adiante. Usa replicação ativa com votação, junto com um protocolo de comunicação de grupo mais robusto, contra falhas que ocorrem quando um objeto ou processador envia uma mensagem que é sintaticamente ou semanticamente incorreta. A replicação de objetos não requer nenhuma modificação nos objetos ou nos métodos.

No Eternal, quando um objeto cliente invoca uma operação a um grupo de objetos servidores que opera segundo a replicação passiva [MOS 98], envia a operação para o grupo dos objetos servidores através do sistema Totem, e uma única réplica servidora (a réplica primária) executa a operação. Cada uma das réplicas armazena a mensagem que contém a invocação, de tal forma que essas réplicas possam invocar a operação, caso a primário falhe. Ao finalizar a operação, o administrador de réplicas atualiza o estado da réplica primária para as réplicas não-primárias, servindo como *checkpoint* no caso de uma falha da réplica primária, e o resultado é enviado para o objeto cliente, via Totem. A transferência de estado serve para ativar a consistência das réplicas. O protocolo de *multicast* confiável de ordem total assegura que todas, ou que nenhuma, das réplicas não-primárias tenham o estado atualizado.

Na replicação ativa do sistema Eternal [MOS 98], a invocação de uma operação, por parte de um objeto cliente, a um grupo de objetos servidores usa o Totem para enviar as operações ao grupo de objetos servidores, onde cada réplica executa a operação. O protocolo de *multicast* confiável de ordem total, neste caso, garante que todas as réplicas recebam as mesmas mensagens na mesma ordem e possam executar as operações seguindo a mesma seqüência, assegurando que o estado das réplicas seja consistente até o fim da operação. Assim como na replicação passiva, também é usado o sistema Totem para enviar os resultados. Mas o papel do administrador de réplicas difere um pouco: ele detecta e suspende as invocações e as respostas duplicadas.

Para tolerar as falhas de omissão, o Eternal usa replicação passiva ou ativa, mas sem efetuar votação. Para tolerar falhas do tipo bizantino[2], deve ser usada replicação ativa com votação por maioria. A votação por maioria requer no mínimo três réplicas ativas, onde as invocações são necessariamente duplicadas. Especialmente, para tolerar falhas arbitrárias, as invocações de réplicas diferentes pertencentes ao mesmo grupo de objetos são controladas e combinadas, usando a votação por maioria produzindo uma única invocação. Através de um processo semelhante é também produzida uma única resposta.

A replicação de objetos é a base na qual são construídos os sistemas distribuídos tolerantes a falhas. Mas esta base só é válida se o estado das réplicas permanecer consistente, até mesmo na presença de falhas.

A fim de que as réplicas de um objeto sejam atualizadas de modo consistente, o Eternal usa o serviço de entrega confiável de mensagens totalmente ordenado, como o

[2] No texto original é usado o termo “commission faults”, que é definido como: ocorre quando um objeto ou processo envia uma mensagem que esta incorreta sintática ou semanticamente, tal como uma mensagem mutante. Mensagens mutantes são duas ou mais mensagens que supostamente deveriam corresponder a mesma mensagem mas têm diferentes conteúdos.

do Totem, que provê sincronismo virtual [MON 98]. A invocação de operações nas réplicas e as respostas correspondentes são contidas em mensagens *multicast.* A garantia de ordem total na entrega das mensagens a todas as réplicas, assim como a garantia de ordem na execução das operações pelas réplicas, mantendo a consistência do estado das réplicas, não é suficiente para manter a consistência dos objetos replicados. O Eternal adiciona alguns mecanismos para manter a replicação consistente como:

- detecção de invocações duplicadas e de respostas duplicadas que são geradas por duas ou mais réplicas de um objeto;

- transferência de estado entre réplicas enquanto o processamento continua assegurando que todas as réplicas concordem em quais operações precedem a transferência de estado e qual será a próxima;

- escalonamento consistente de operações concorrentes, permitindo operações locais sem requerer comunicação da rede;

- realização de operações para reconciliar inconsistências resultantes da continuidade das operações em situações de particionamento da rede e uniões subseqüentes.

2.4 Análise sobre as abordagens

A análise aqui mostrada representa [FEL 98a] uma visão geral sobre as diferentes abordagens, enfocando diferentes aspectos como: transparência, facilidade de uso, portabilidade, interoperabilidade e modularidade. Foi baseada nas implementações de algumas ferramentas como o Electra [FEL 96, FEL 98a], Orbix+Isis [FEL 96, FEL 98a], Eternal e o OGS [FEL 98a].

A **transparência** possibilita ao programador abstrair o uso do grupo de objetos, ou seja, lhe dá a ilusão de que as invocações são emitidas e recebidas de objetos simples. A abordagem de integração provê transparência do lado do cliente: o invocador não precisa saber que o invocador é um grupo, embora ele possa beneficiar-se deste conhecimento. Sabendo que está invocando um grupo de objetos, pode obter todas as respostas e não apenas uma de forma transparente. Já na abordagem de interceptação, a transparência ocorre de forma obrigatória, isto é, um cliente não tem acesso a todas as respostas de uma invocação de *multicast*. A abordagem de serviço pode ser configurada com ou sem transparência, dependendo do tipo de serviço especificado.

Uma consideração importante é **facilidade de uso**, a qual proporciona agilidade no desenvolvimento de aplicações e pode tornar a aplicação mais robusta e confiável, reduzindo as chances de erros dos programadores. O suporte à transparência de grupo das abordagens de integração e interceptação facilita o projeto do cliente, pois não requer qualquer construção explícita no lado do cliente. O cliente não diferencia se o servidor corresponde a um grupo ou não. Por outro lado, utilizar as vantagens da comunicação de grupo como, por exemplo, associar uma semântica particular com uma invocação de *multicast*, é mais complexo com enfoques transparentes, pois não é ortogonal aos grupos de invocação. A abordagem de serviço também apresenta uma boa facilidade de uso na sua forma configurável.

A **portabilidade** representa o quanto um componente de *software* é independente da camada básica de comunicação ou arquitetura. Felber observou dois tipos de portabilidade: no código de suporte a grupos e a portabilidade do código da aplicação. A abordagem de integração apresenta o seu código de suporte a grupos integrados na camada básica de comunicação, tornando-se totalmente não portável para outras arquiteturas; adicionalmente, a utilização de linguagens específicas não padronizadas pode tomar o código da aplicação não portável. Com a interceptação, o código de suporte a grupo não é portável para quaisquer arquiteturas, pois usa características de baixo nível do sistema operacional; mas o código da aplicação é independente do mecanismo de interceptação, sendo assim completamente portável. Ao contrário das abordagens que utilizam um conjunto de ferramentas de comunicação de grupo, na abordagem de serviço, tanto o código do suporte a grupo como o da aplicação são portáveis.

A **modularidade** permite a redução da complexidade de um sistema, subdividindo-o em um conjunto de componentes coesos, mas fracamente acoplados livremente. Neste caso, cada componente pode ser modificado, estendido ou replicado sem requerer qualquer mudança em outros componentes, tornando o sistema assim mais fácil para sofrer manutenção de *software*. É difícil avaliar a modularidade em uma abordagem devido ao fato de que envolve considerações de implementação. Porém, desta forma, uma regra geral é que componentes isolados são mais modulares que um monolítico. A abordagem de integração provê uma arquitetura monolítica, pois tem o suporte a grupo como parte da camada básica de comunicação. Assim sendo, ela apresenta uma modularidade limitada. Por outro lado, a interceptação tem o suporte a grupo separado, preservando a modularidade. Já a abordagem de serviço provê a reutilização e a modularidade para a definição de novos componentes, os quais podem ser integrados com outros serviços ou aplicações.

O termo **interoperabilidade** aparece definido na tese de Felber [FEL 98a] vinculado ao conceito de CORBA. Entretanto, uma adaptação da definição a um ambiente genérico poderia ser formulada da seguinte maneira: Interoperabilidade é a propriedade de interação entre diversas aplicações que executam sobre ambientes com multiplicidade de implementações. Esses ambientes devem definir um protocolo padrão para prover a interoperabilidade. Assim, as implementações de aplicações que usarem esse protocolo podem interoperar umas com as outras, tornando fácil a integração de componentes heterogêneos de fabricantes diversos.

As implementações baseadas em ferramentas de grupo ou protocolos patenteados (ou de código privativo aos fabricantes) tendem a não ser interoperáveis, que é o caso das abordagens de integração e de interceptação. No caso da abordagem de serviço, ocorre o uso somente das primitivas de comunicação que estão por baixo da camada básica de comunicação, sendo assim, completamente interoperável.

A interoperabilidade é um aspecto que não foi considerado no presente estudo, pois se entendeu que a interoperabilidade está ligada diretamente com a capacidade da ferramenta ser independente quanto ao sistema de comunicação, ou seja, ser portável para diferentes sistemas de comunicação. Observa-se que o escopo deste trabalho tem

como objetivo estudar as abordagens de captura das mensagens e não o sistema como um todo.

3 Estudo de casos

Neste capítulo, encontra-se o estudo de alguns projetos que empregaram as abordagens de interceptação, serviço e integração, com a finalidade de prover as mais variadas funcionalidades. Na figura 3.1, apresenta-se uma classificação dos projetos que tem por base as abordagens empregadas na implementação destes. A montagem da tabela atende os seguintes critérios: (i) a classificação apresentada por Felber [FEL 98], que subdivide os enfoques de implementação em três abordagens básicas: interceptação, serviço, e integração; (ii) o segundo critério usado, refere-se ao mecanismo usado para prover esta abordagem, cuja subdivisão foi apresentada por Lung [LUN 2000a]. Observa-se que este segundo critério aplica-se apenas à integração.

A abordagem de interceptação prevê que a captura das informações possa ser suportada por três meios diferentes. Para tal, podem ser usadas: a estrutura da linguagem, a interface do sistema operacional, ou mesmo o mecanismo do próprio ORB. A figura 3.1 mostra alguns projetos que fazem uso destes diferentes meios. Neste capítulo, também são apresentados os estudos de casos destes projetos, onde são abordados de modo mais completo estas três formas de captura das informações da abordagem de interceptação.

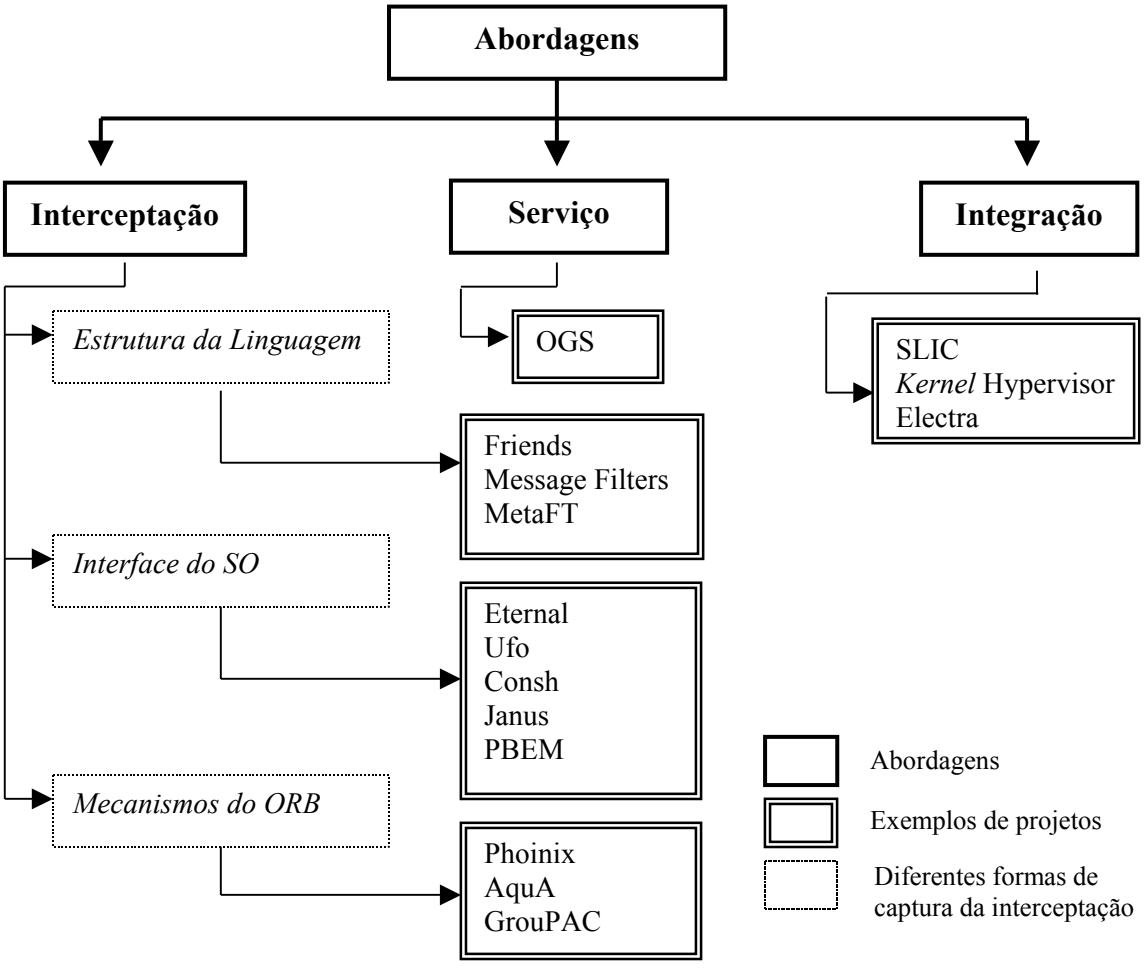


FIGURA 3.1 – Classificação dos projetos segundo as abordagens

3.1 FRIENDS

O sistema FRIENDS (*Flexible and Reusable Implementation Environment for your Next Dependable System*) [FEL 96, FAB 98] foi desenvolvido utilizando uma arquitetura de meta níveis composta de sub-sistemas e bibliotecas de classes de meta-objetos para prover tolerância a falhas, comunicação segura e aplicações distribuídas baseadas em grupo. O uso de meta-objetos proporciona uma boa separação de interesses entre mecanismos e aplicação. Os meta-objetos podem ser utilizados de forma transparente pela aplicação e permitem que a sua composição seja de acordo com as necessidades de uma determinada aplicação; a flexibilidade é incrementada pela reusabilidade dos meta-objetos.

Protocolos de meta-objetos são baseados em reflexão e orientação a objetos. A reflexão expõe à linguagem de programação um alto nível de abstração, tornando isto compreensível enquanto preserva a eficiência e a portabilidade da linguagem de programação. Orientação a objetos oferece uma interface para linguagens de programação na forma de classes e métodos, assim variantes das linguagens-padrão podem ser criadas.

As instâncias destas classes são chamadas de meta objetos. Em um protocolo de meta-objetos simples, o meta-objeto é projetado para executar algumas ações quando o objeto é criado, destruído ou quando um método do objeto é invocado. O meta-objeto também pode ativar métodos e acessar atributos do objeto, ou seja, acessar o estado do objeto. O FRIENDS beneficia-se das propriedades oferecidas pelo MOP (*metaobject protocols*), através da utilização da extensão da linguagem C++, o Open C++ [CHI 93, CHI 95], com a finalidade de controlar o comportamento dos objetos da aplicação.

A arquitetura é composta por diferentes camadas, como mostra a figura 3.2; (i) a camada de *kernel* que pode ser um *kernel* de Unix ou preferencialmente um *micro-kernel* como Chorus; (ii) a camada do sistema composta por diferentes subsistemas dedicados, um para cada abstração; (iii) a camada do usuário dedicada à implementação de aplicações e mecanismos de meta-objetos.

A implementação de qualquer abstração como tolerância a falhas, comunicação segura ou distribuição é dividida entre as bibliotecas de classes de meta-objetos e o subsistema correspondente.

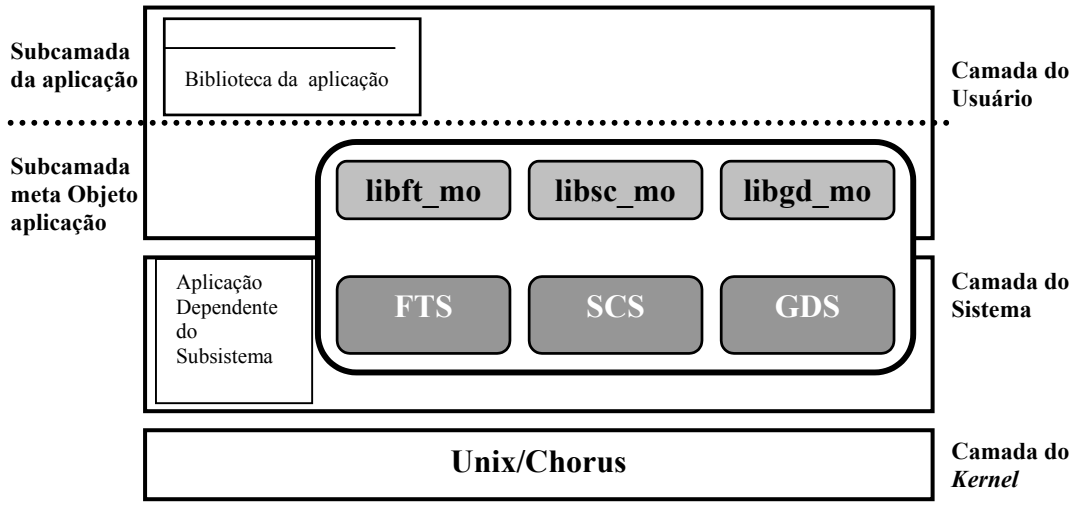


FIGURA 3.2 - Arquitetura do FRIENDS.

A composição da arquitetura disposta em camadas é organizada como um conjunto de subsistemas. Em tecnologia de *micro-kernel* um subsistema corresponde a um conjunto de serviços que implementam qualquer sistema de *software*, por exemplo, um sistema operacional sobre um micro-*kernel* (Unix ou Chorus). No Friends, cada subsistema provê um serviço como tolerância a falhas ou comunicação segura ou distribuição baseada em grupo, estes subsistemas na camada do sistema são os seguintes:

- FTS (*Fault Tolerance Sub-system*) - provê serviços básicos obrigatórios para computação tolerante a falhas, em particular detectores de defeitos, os quais devem ser implementados em baixo nível. Este subsistema também engloba configuração e facilidades de gerência de replicação e oferece suporte para armazenamento estável.

- SCS (*Secure Communication Sub-system*) - provê serviços básicos que devem ser implementados com uma entidade confiável dentro do sistema. Este serviço deveria incluir em particular um servidor de autenticação, mas também um servidor de autorização, um servidor de diretório e um servidor de auditoria.

- GDS (*Group-based Distribution Sub-system*) - provê serviços básicos para implementar um suporte de distribuição para aplicações orientadas a objetos, onde estes podem ser replicados. Estes serviços básicos incluem facilidades de administração de grupo e protocolo de *multicast* atômico.

A camada do usuário é dividida em duas subcamadas: a camada da aplicação e a camada de meta-objetos, que controla o comportamento dos objetos da aplicação. Constituem-se de algumas bibliotecas de classes de meta objetos para a implementação de tolerância a falhas. As aplicações distribuídas seguras são implementadas em cima do sub-sistema correspondente e proporcionam ao usuário mecanismos que podem ser ajustados usando técnicas de orientação a objeto são elas:

- libft_mo - proporciona classes de meta objetos para várias estratégias de tolerância a falhas, baseadas em armazenamento estável ou replicação, visando tolerar falhas físicas que se refletem em nodos *fail-silent*.

- libsc_mo - provê classes de meta objetos para diversos protocolos de comunicação segura usando técnicas de cifragem, assinatura e verificação baseados em chaves secretas e públicas ou sistema de criptografia.

- libgd_mo - provê meta objetos para gerenciar interação de objetos remotos que podem ser implementados com grupos. A combinação destes meta-objetos e do GDS proporciona um suporte, em tempo de execução, para aplicações distribuídas orientadas a objetos.

3.2 Message Filters for Object-oriented Systems

No modelo convencional de objetos, objetos encapsulados interagem através de mensagens que resultam em invocações de métodos no objeto destino. Uma mensagem é entregue diretamente ao objeto destino, como resultado destas entregas diretas. O código de controle da mensagem é executado de modo intermediário, tratando a mensagem e não pode ser abstraído ou separado do código de processamento da mensagem no objeto destino, sem sacrificar a transparência do controle intermediário da mensagem [RUS 97].

O modelo de filtro de passagem de mensagens para linguagens orientadas a objetos é proposto para prover a separação entre o controle e o processamento da mensagem de uma forma transparente. Um relacionamento interclasses, chamado *filter relationship*, é introduzido. Como conseqüência, um objeto-filtro pode interceptar e tratar mensagens, enviando mensagens para outros objetos chamados clientes, via funções dos filtros-membros. O modelo de filtro suporta os mecanismos de filtragem para mensagens ascendentes e descendentes, facilitando a interceptação de mensagens enviadas e valores retornados [RUS 97]. O filtro pode ser conectado e desconectado em tempo de execução.

A reação de uma mensagem pode ser dividida em dois estágios: o controle e o processamento da mensagem. O código de controle da mensagem é o código que é executado de modo intermediário, tratando a mensagem antes desta ser entregue ao objeto-destino. O código de processamento da mensagem é o código que atua na destinação do objeto que processa a mensagem para ativar as funcionalidades desejadas pelo objeto que fez a requisição. Separar o controle das mensagens de seu processamento pode ser muito vantajoso. A separação cria a possibilidade de desenvolver uma política de controle de mensagens em uma forma modular [RUS 97]; esta flexibilidade não é possível no modelo de entrega direta, pois controle e processamento são acoplados.

O modelo de entrega filtrada (*filter delivery*) - que pode ser visto como um novo modelo de comunicação de mensagens interprocessos - provê uma forma modular para desenvolver objetos que agem como filtros de mensagens para seus objetos-cliente. Os objetos-filtro agem transparentemente, removendo, adicionando ou replicando. Os

objetos-filtro não requerem qualquer modificação no código dos objetos-origem ou destino.

O modelo de filtro proposto por Rushikesh [RUS 97] é similar ao mecanismo de filtro proposto no Orbix CORBA. O Orbix permite aos programadores o fornecimento de filtros para clientes e principalmente para servidores para empacotar requisições como autenticações, estatísticas de desempenho, auditoria e codificação. O principal propósito de mecanismos de filtro, como o implementado no Orbix, é o de sustentar flexibilidade e leveza. O modelo de filtro é provido ao nível de linguagem introduzindo um filtro de relacionamento (*filter relationship*) entre classes. As principais características deste modelo incluem a criação dinâmica de filtros de relacionamento entre objetos; por mensagens e filtragem seletiva e troca dinâmica das políticas de filtragem.

Alguns padrões de projetos como o *decorator* e *proxy* provêem, de um certo modo, a funcionalidade do modelo de filtro de entrega. Porém, os filtros introduzidos aqui são programados em nível de linguagem, e a principal intenção é para unir objetos clientes e servidores, usando um objeto-filtro de uma forma transparente [RUS 97]. Padrões de projeto são feitos em nível da estruturação de sistemas orientados a objetos; é possível construir padrões de projeto específicos baseados no modelo de objetos-filtro.

O objeto-filtro pode fazer as seguintes ações referentes à interceptação de mensagens:

- interceptação de mensagens ascendentes (*interception of upward messages*): mensagens ascendentes são mensagens de um objeto-origem para um objeto-destino. As funções que filtram estas mensagens são chamadas de *upfilter*.

- Manipulação de mensagens (*manipulation of messages*): um *upfilter* poderá trocar o argumento de uma mensagem e processar em um código arbitrário.

- Devolução (*bounce*[3]): um *upfilter* retorna um valor para a origem da mensagem em lugar do destinatário.

- Transmissor (*pass*): um *upfilter* transmite a mensagem para o filtro-cliente após uma possível manipulação das constantes da mensagem.

- Intermediação das invocações em outros objetos (*intermediate invocation on other objects*): um objeto-filtro pode enviar requisições para outros objetos como parte de seu código de controle.

- Interceptação de mensagens descendentes (*Interception of downward messages*): mensagens descendentes são os valores que retornam. Uma mensagem descendente pode também ser filtrada. A função que filtra uma mensagem descendente é denominada de *downfilter*.

[3] *Bounce* é um retorno realizado pelo filtro. A ação de *pass* consiste no repasse ao cliente.

Várias propriedades de um objeto-filtro podem ser descritas do ponto de vista de suas funcionalidade e uso. Essas propriedades foram divididas em duas categorias: essenciais e estendidas [RUS 97]. A categoria essencial concentra-se em funcionalidades básicas do objeto-filtro. As propriedades estendidas exploram a flexibilidade do ponto de vista do uso.

As propriedades essenciais para objetos-filtro são definidas como:

- suporte para ações básicas de filtragem: objetos-filtro são especificados para interceptar mensagens que chegam a um objeto normal chamado um *filter-client*. Filtros podem manipular as mensagens. Eles transmitem ou retornam (*bounce*) essas mensagens, em tempo finito (*eventually*). Esta propriedade provê construções elegantes de linguagem para capacitar o projeto de objetos que executam tarefas relacionadas ao controle de mensagens como a verificação de limites, segurança, conversões de dados, processamento de mensagens e roteamento de mensagens.

- Modularidade: a especificação de um objeto-filtro é separada da especificação de um objeto-cliente. O objeto-filtro não rompe o encapsulamento do seu cliente, nem o cliente rompe o encapsulamento do seu filtro.

- Transparência: o transmissor não precisa saber da existência do objeto-filtro, preservando a semântica de chamada direta para o remetente da mensagem. Assim, não há alterações no código-fonte do objeto-origem desta requisição, quando um objeto-filtro é adicionado, removido ou replicado.

- Filtragem seletiva: filtros podem interceptar mensagens seletivamente. Alguns dos métodos podem ficar sem serem filtrados, enquanto que outros podem ser filtrados. Essa propriedade permite que um objeto-filtro implemente códigos independentes de controle de mensagens para múltiplas mensagens.

As propriedades estendidas são definidas como:

- Grupo de filtragem: permite que vários *filter-client* possam ser servidos por um simples filtro. Essa propriedade define uma extensão óbvia do poder de um objeto-filtro. Um objeto-filtro pode interceptar mensagens enviadas para um número de objetos-clientes que são instâncias da mesma classe-cliente.

- Filtragem dinâmica: filtros podem ser trocados por um cliente durante suas vida. Essa propriedade descreve a capacidade de ligação dinâmica dos filtros. A ligação é feita em dois níveis: no primeiro, o objeto-filtro pode ser removido e replicado. No segundo nível, funções de membros individuais no interior do objeto-filtro, que filtram suas atividades correspondentes nos objetos-clientes, podem ser trocadas em tempo real.

- Filtragem de camada: essa propriedade descreve que filtros podem ser aninhados. Filtros multiníveis podem ser usados para projetar o

processamento de mensagens multiníveis. Por exemplo, um filtro pode encarregar-se da segurança, enquanto outro pode ser anexado para funcionar como um roteador para redirecionar mensagens para outro servidor.

Através dessas propriedades, é possível perceber que o modelo de objeto-filtro básico oferece suporte para relacionamento de um-para-um, isto é, de um objeto-filtro e um objeto-cliente do filtro. No modelo estendido, o suporte é de um-para-vários, através da filtragem de grupo, e de vários-para-um com os filtros de camada.

Uma classe filtro é organizada da seguinte forma. Ela define uma interface chamada de interface-filtro, separada em interfaces privadas e públicas e ambas são independentes [RUS 97]. Uma interface-filtro define funções-membro que são invocadas automaticamente quando as funções-membro correspondentes no objeto-cliente são interceptadas pelo objeto-filtro. Assim, os membros da interface-filtro são invocados apenas pelo sistema que executa o programa. As funções membro definidas na interface-filtro não são acessíveis como membros privados ou públicos.

3.3 MetaFT

O modelo MetaFT adota uma estrutura reflexiva segundo a abordagem de meta-objetos. A implementação das técnicas de replicação é também apoiada no uso de ORBs com suporte para grupo de objetos em ambientes heterogêneos. O modelo MetaFT aumenta a flexibilidade permitindo, por exemplo, que se mude os protocolos de coordenação de réplicas, mudando de técnica, sem qualquer implicação para o código da aplicação [LUN 99b].

A essência do paradigma de reflexão computacional está no fato de que um sistema pode executar processamento sobre si mesmo, modificando então o seu comportamento [LUN 97]. O paradigma reflexivo é introduzido na programação orientada a objetos na forma da abordagem de meta-objetos, onde os aspectos funcionais e não funcionais são separados com o uso de objetos-base e meta-objetos. Os objetos-base descrevem com os seus métodos as funcionalidades da aplicação, enquanto os meta-objetos associados executam as políticas de controle que determinam o comportamento de seus objetos-base correspondentes. As chamadas aos métodos de objetos-base são desviadas no sentido de ativar meta-métodos, que permitem modificar o comportamento de objetos-base ou adicionar funcionalidades às correspondentes chamadas em nível base [LUN 99b].

A computação reflexiva permite atribuir a objetos-base as funcionalidades da aplicação replicada, enquanto meta-objetos executam os protocolos de coordenação entre réplicas [LUN 99b]. Isto possibilita a utilização de diferentes técnicas de replicação com os objetos-base mantendo suas características, bastando para isso trocar os meta-objetos associados.

A figura 3.3, mostra a estrutura reflexiva proposta para incorporar os conceitos de técnicas de replicação. Cada réplica foi mapeada sob a forma de um objeto-base, ao qual se associou um meta-objeto que assume funções de coordenação da técnica usada. Nos modelos implementados, foi assumida a hipótese de falhas de colapso (ou *crash*) [LUN 99b]. Como é admitido um acoplamento forte entre o controlador e a réplica

associada, os erros gerados serão atribuídos a ambos; na hipótese de colapso, o controlador e a réplica associados cessarão de executar.

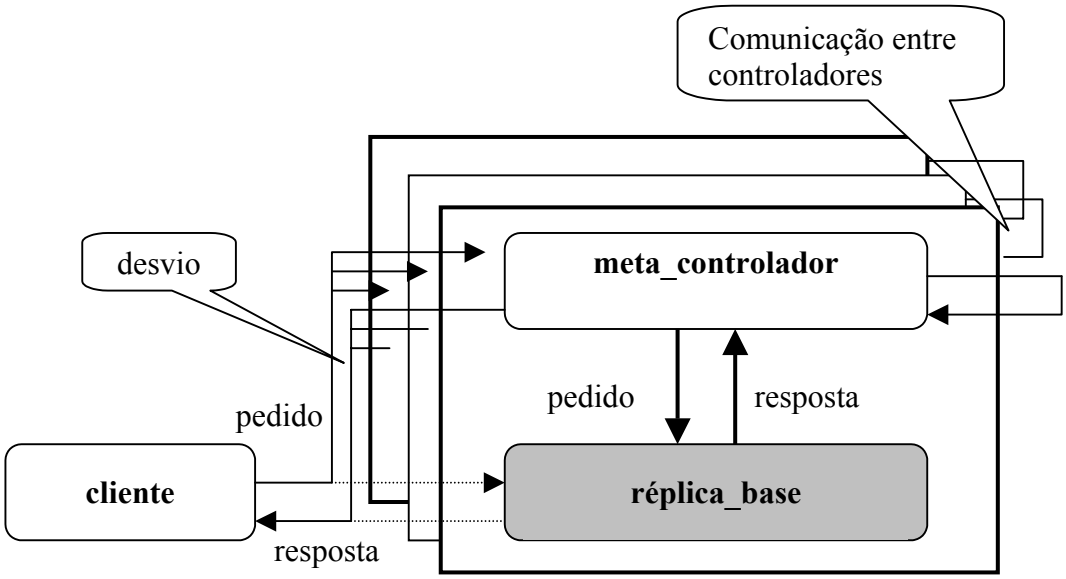
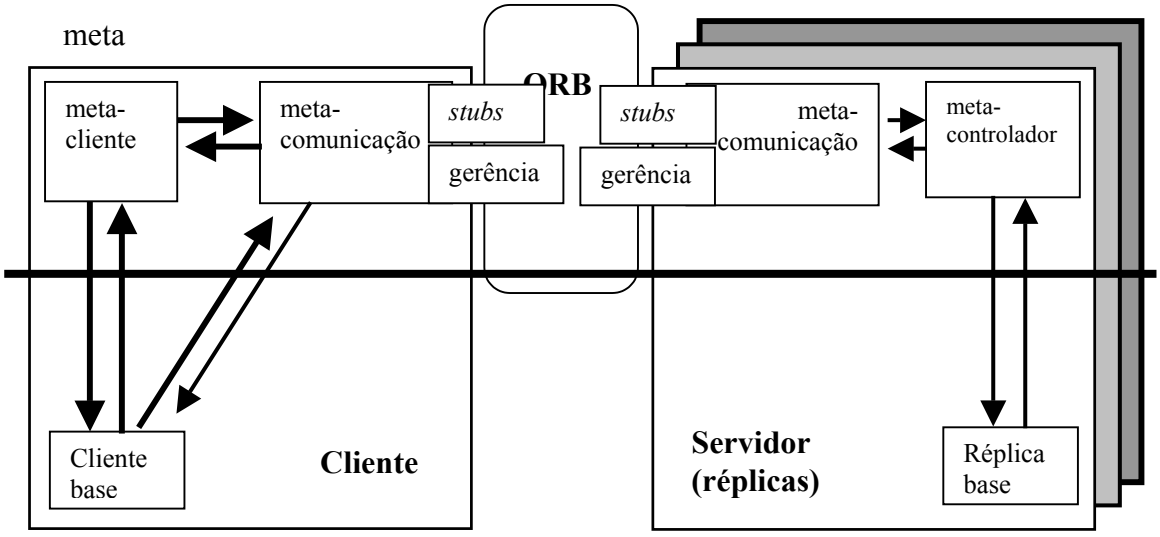


FIGURA 3.3 - Estrutura reflexiva para o modelo de replicação.

O modelo de integração de técnicas de replicação no contexto CORBA é mostrado na figura 3.4. Nesta figura, o cliente apresenta-se estruturado em um cliente-base que representa o comportamento da aplicação, e um meta-cliente, que não possui função ativa em nesta implementação, mas que poderia ser usado na gerência de um cliente replicado, ou para implementar mecanismos de tratamento de exceções no cliente [LUN 99b]. A estrutura de cada réplica do servidor é semelhante a do cliente: um objeto-réplica base, realizando o serviço que se deseja replicar; um meta-controlador, responsável pelo protocolo de coordenação da técnica de replicação; e meta-objetos especiais, identificados genericamente como meta-comunicação, que controlam tanto no lado do cliente como no lado do servidor o acesso ao suporte fornecido por uma plataforma CORBA.



base

FIGURA 3.4 - Estrutura do modelo sobre um suporte CORBA.

Este modelo pode ser usado com várias técnicas de replicação, as diferenças essencialmente se concentrarão nos meta-controladores do servidor replicado. Em algumas técnicas, as entidades de meta-comunicação podem ganhar funcionalidades além daquelas de concentrar métodos de acesso aos serviços do suporte CORBA. Por exemplo, no uso de réplicas ativas com mecanismos de votador ou ajustador, a implementação da votação ou ajuste é programada de maneira menos custosa no lado do cliente. A transparência pode ser conseguida neste caso, implementando esses mecanismos na entidade meta-comunicação do cliente que, com a adição dessa funcionalidade, ganha as características de um objeto real [LUN 97].

3.4 Ufo e Consh

A extensibilidade é um assunto conveniente e importante em sistemas operacionais modernos. Sistemas operacionais extensíveis podem ser facilmente ajustados para satisfazer a requisitos de novas aplicações. O trabalho de Albert Alexandrov [ALE 99] mostra como estender um sistema operacional, completamente em nível de usuário, sem modificar o próprio sistema: ele é baseado na interposição de chamadas de sistemas. A utilização da interceptação selecionada de chamadas de sistema em nível de usuário, adotada neste trabalho, beneficia-se de facilidades disponibilizadas por interfaces de monitoramento como o sistema de arquivos `/proc` e a chamada de sistema *ptrace*, esta última descrita na seção 4.1.5. Estas interfaces encontram-se disponíveis na maioria dos sistemas operacionais baseados em Unix.

As interfaces de monitoramento possibilitam que os comportamentos de algumas chamadas de sistema interceptados sejam modificados para a implementação de novas funcionalidades. Esta abordagem não requer que sejam realizadas qualquer nova ligação (*linkage*) nem re-compilação da aplicação existente. Na realidade, as extensões podem até mesmo ser "instaladas" dinamicamente sobre um processo já em execução. As extensões trabalham completamente em nível de usuário e podem ser instaladas sem a necessidade de um administrador de sistema. O usuário poder individualmente escolher que extensão será executada, podendo criar uma visão personalizada do sistema operacional para ele. Alexandrov demonstrou a viabilidade destas aproximações através da implementação de duas extensões do sistema operacional: Ufo e Consh. **Ufo** é um sistema de arquivos globais que provê acesso remoto transparente para serviços HTTP e FTP. **Consh** é semelhante a um *shell* confiável para execuções protegidas que oferecem serviços para novas aplicações da Internet.

3.4.1 Ufo

O **Ufo** [ALE 98a, ALE 99] é uma ferramenta baseada em um interceptador (*catcher*); estende o sistema operacional padrão Unix, o Solaris 2.5.1[4], completamente em nível de usuário. Nele é implementado um sistema global de arquivos que permite às aplicações locais a acesso a arquivos em máquinas remotas de modo transparente. Ele é um processo em nível de usuário que executa sobre o sistema Unix; conecta-se às máquinas remotas via protocolos FTP e HTTP com autenticação ou de forma anônima [ALE 97] e também oferece escrita e leitura com uma política de consistência fraca.

[4] Solaris é marca registrada da Sun Microsystems, cuja página virtual pode ser encontrada em http://www.sun.com.br/ ou http://www.sun.com.

A sua arquitetura é dividida em dois módulos: o interceptador e o módulo Ufo, conforme mostrado na figura 5.1. O interceptador tem a responsabilidade de capturar as chamas de sistema e enviá-las para o módulo Ufo. Neste segundo módulo, é implementado o sistema de arquivos remoto que consiste em três camadas: a camada de arquivos de serviço, os quais identificam os arquivos remotos, a camada de memória (*caching*), e a camada de protocolos que contém os diferentes módulos do tipo *plug-in* com os protocolos de transferência atuais.

A figura 3.5 ilustra como o Ufo trata as chamadas de sistemas. Quando uma aplicação faz uma chamada de sistema (1), pode ir diretamente para o *kernel* ou, se é relacionada com arquivo, é capturada pelo interceptador (2) e repassada para o Ufo. Para capturar as chamadas, o Ufo determina se a chamada de sistema opera em um arquivo remoto ou local, possibilitando usar os serviços do *kernel* (3). Se o arquivo é local, o procedimento de requisição não é modificado. Se o arquivo é remoto, o Ufo invoca mais serviços do *kernel* (4) para criar uma cópia local, e o *patch* da chamada é modificado. Finalmente, o Ufo passa a requisição para o *kernel* (5). Após o serviço ser requisitado no *kernel* (6), o resultado pode ser retornado diretamente para aplicação (10), ou pode ser novamente capturado pelo interceptador do Ufo (7, 8, 9).

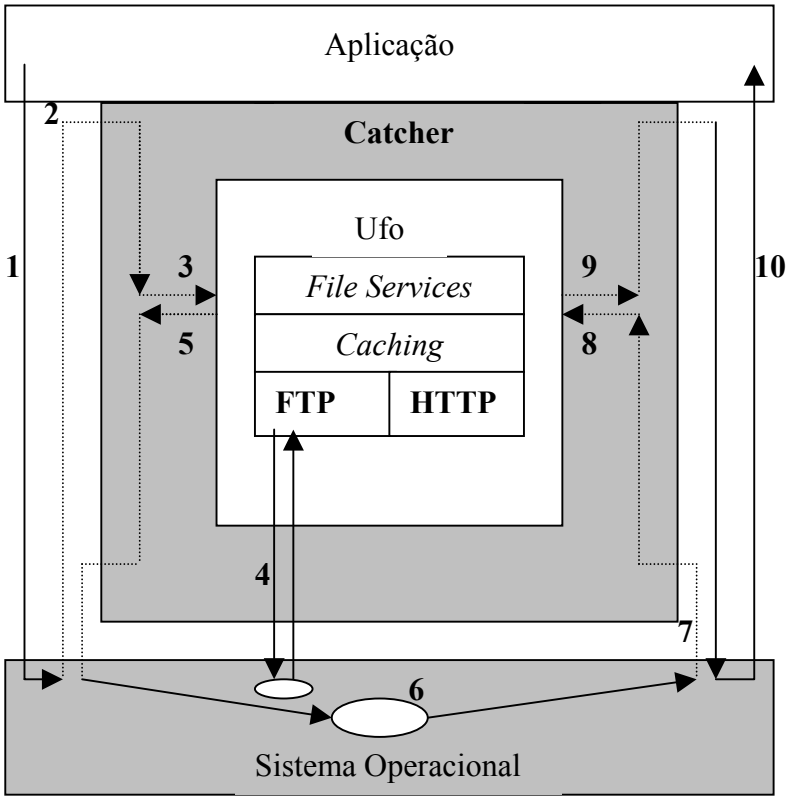


FIGURA 3.5- Arquitetura Geral do Ufo

O interceptador foi implementado utilizando o sistema de arquivo virtual `/proc` do Solaris, mecanismo também usado por programas depuradores como o *truss* ou *strace*. O interceptador prende-se a um processo dono de um *pid,* abrindo o seu respectivo arquivo no diretório `/proc/pid`. Uma vez preso, o interceptador usa as chamadas de sistema *ioctl* para abrir o descritor de arquivo para controlar o processo

[ALE 98a, ALE 99]. Ele pode instruir o sistema operacional a parar o processo preso em numerosos eventos de seu interesse. No Ufo, são dois os eventos de interesse: a entrada e a saída das chamadas de sistema do *kernel*. Ele usa os eventos de interesse para examinar e modificar os parâmetros ou resultados de chamadas de sistema como *open, stat, e getdents* [ALE 97]. Finalmente, a interface `/proc` possibilita o reinício da execução do processo parado. Eventualmente, o processo pode chegar ao fim de sua execução ou ser morto: o interceptador detecta isso e pára o monitoramento do processo.

O Ufo suporta três formas de especificar nomes de arquivos remotos: (i) através de uma URL, (ii) através de um nome de arquivo implícito contendo o *host* remoto, nome do usuário, e modo de acesso, (iii) através de um ponto de montagem *(mountpoint).*

3.4.2 Consh

Consh é um ambiente confiável para computação na Internet. Ele pode executar binário confiável sob um ambiente controlado, verificando os acessos aos recursos da máquina e potencialmente negando acessos perigosos. O mecanismo de proteção do Consh não é uma idéia nova [ALE 98b]. A idéia de proteção do Consh está diretamente baseada em outro projeto chamado Janus, o qual é descrito na seção 3.5. O Consh incorpora uma versão modificada do Janus em um de seus módulos. A diferença entre o Consh e os esquemas de proteção do Janus é que o Consh combina proteção para acesso local e alternativamente ou adicionalmente recursos remotos. O Consh virtualiza todos os recursos visíveis para a aplicação, permitindo recursos de proteção tais como a replicação transparente de arquivos com alternativa de segurança local ou remota.

Ele restringe os binários confiáveis, mas também pode oferecer-lhes novas capacidades através de benefícios tal como simplificar os projetos de aplicações para a Internet [ALE 99, ALE 98b]. Por exemplo, em aplicações de computação global, as diferentes partes da computação podem acessar transparentemente recursos remotos providos pelo Consh tais como arquivos de dados e executáveis, sem qualquer esforço de programação adicional. Um outro exemplo é o caso em que a aplicação precise ter acesso a um arquivo que o usuário deseja manter protegido; o Consh provê meios para tal, sem relegar qualquer das garantias de proteção.

O Consh não é um *shell* Unix como o nome sugere; ele é um programa executado como um processo ao nível de usuário [ALE 99]. O processo Consh pode anexar outro processo e então controlá-lo. Em particular, qualquer *shell* Unix pode ser executado como um processo anexado sob o Consh. Ele é constituído de três módulos: o interceptador (*Catcher*), recursos virtuais (VR), o qual é baseado no Ufo, e o módulo Janus. A implementação do Consh é para Solaris 2.5 em C++ usando o sistema de arquivos `/proc` para interceptar chamadas de sistema. Utiliza o Ufo para prover acesso remoto transparente a sistemas de arquivos FTP e HTTP e o Janus, que é um ambiente delimitador ao nível do usuário, isto é, auxilia a restringir o acesso da aplicação a recursos locais, restringindo a rede e controlando o ambiente com melhor segurança [ALE 98b].

A arquitetura do Consh está baseada em uma infra-estrutura construída para interpor chamadas de sistema (SCI). A idéia por trás do SCI é que, em aplicações que se comunicam com o sistema operacional usando chamadas de sistema, se for inserida uma camada entre as aplicações e o sistema operacional que capture as chamadas de sistema e modifique o comportamento delas, pode-se conseqüentemente alterar ou estender as funcionalidades do sistema operacional de modo que as aplicações entendam [ALE 98b]. Esta infra-estrutura consiste de um interceptador e extensões opcionais, como ilustrado na figura 5.3. O interceptador é o componente central que captura as chamadas e as passa para os módulos. No Consh, as extensões de módulo são: Ufo, que provê a extensão do sistema de arquivos e funcionalidade de rede, e o Janus que provê proteção.

O interceptador é capaz de conectar-se a um processo de usuário e capturar as chamadas de sistema emitidas pelos processos. Ele provê uma camada entre os processos da aplicação e o sistema operacional original, conforme ilustrado na figura 3.6. Essa camada extra não exige a troca do sistema operacional, mas permite um controle sobre o ambiente do usuário, por modificação de parâmetros das chamadas de sistema ou retirando as requisições de serviços adicionais ao sistema. O interceptador trabalha da seguinte forma: inicialmente, conecta-se ao processo do usuário e informa ao sistema operacional que está capturando as chamadas de sistema desse processo. Como conseqüência, ele pode trocar ao nível de usuário a semântica das chamadas, capturadas estendendo as funcionalidades do sistema operacional. Sempre que uma chamada de sistema de interesse é iniciada, o sistema operacional pára o processo anexado e notifica o interceptador. O interceptador chama o módulo apropriado, se necessário, e então a chamada de sistema é retornada.

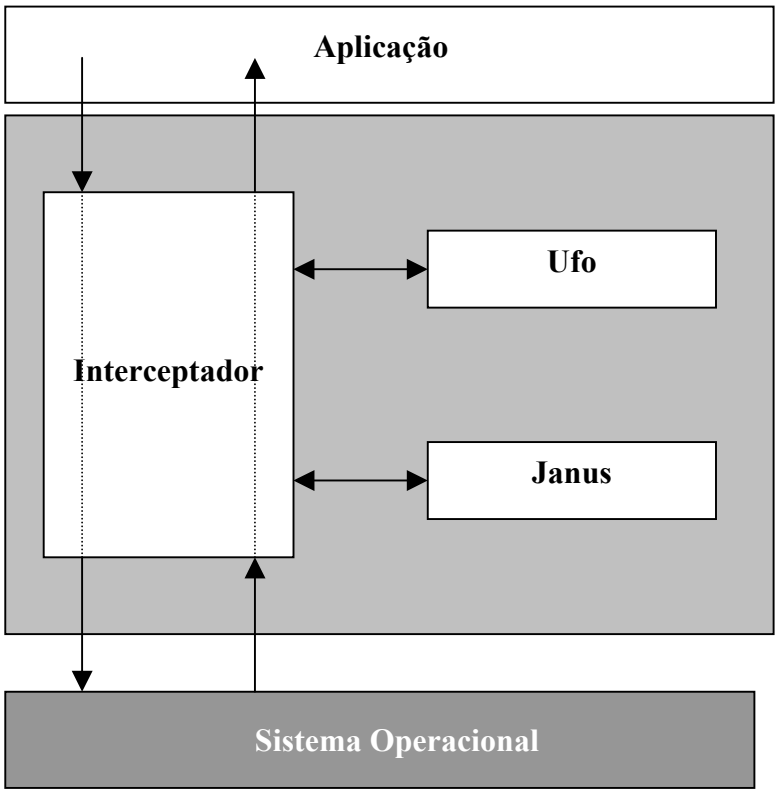


FIGURA 3.6 - Arquitetura do Consh

3.5 Janus

O objetivo do projeto Janus [WAG 99] é oferecer um mecanismo em nível de usuário que monitore aplicações não confiáveis, não permitindo chamadas do sistema que possam causar danos ao sistema. Este projeto tem como metas básicas:

- Segurança - aplicações não confiáveis devem ter impedido o acesso a qualquer parte do sistema ou rede para o qual o Janus não conceda permissão explícita. Ele usa o termo *sandboxing* para descrever o conceito de confinamento de uma aplicação não confiável por um ambiente restrito.

- Versatilidade - possibilita negar ou permitir a execução de uma determinada chamada de sistema. Por exemplo, a chamada de sistema *open* poderia ser permitida ou negada dependendo das permissões associadas ao arquivo. Com isso, mantém-se a flexibilidade, permitindo usar a ferramenta para proteger vários tipos de aplicações.

- Configurabilidade - Diferentes *sites* têm diferentes requisitos, como: a quais arquivos a aplicação deve ter acesso, ou para quais *hosts* a aplicação deve ter permissão para abrir uma conexão TCP. Na realidade, o Janus deve ser configurável deste modo, tanto se opera com base por usuário ou por aplicação.

O Janus adotou um projeto modular: um *framework,* que é a parte essencial, e módulos dinâmicos, usados para implementar vários aspectos de uma política de segurança configurável para filtrar as chamadas de sistema [WAG 99]. Esta decomposição permite separar os mecanismos por política. O *framework* lê o arquivo de configuração, o qual pode ser dependente do usuário, da aplicação, ou do *site*. Cada módulo filtra a saída de uma chamada de sistema invocada. Quando a aplicação tenta uma chamada de sistema, o *framework* despacha as informações relevantes para a política do módulo. Cada módulo relata a sua opinião, isto é, se a chamada de sistema deve ser permitida ou cancelada, e qualquer ação necessária é feita pelo *framework.* Ele segue o princípio do menor privilégio: permite que o sistema operacional execute uma chamada de sistema somente se algum módulo permitir explicitamente; a opção padrão (*default*) define que as chamadas de sistema devem ser negadas.

Cada módulo contém uma lista das chamadas que podem ser examinadas e filtradas, a mesma chamada de sistema pode aparecer em listas de módulos diferentes. Um módulo pode determinar, para cada chamada, uma função arbitrária que valide os argumentos da chamada antes dela ser executada pelo sistema operacional [WAG 99]. A função pode então usar estas informações para opcionalmente atualizar o estado local, e então sugerir a permissão (*allow*) ou negação (*deny*) da chamada, ou não fazer comentário (*no comment*); esta última ocorre quando o módulo não tem nenhuma entrada sobre a chamada.

A sua implementação usa mecanismos diferentes para os sistemas operacionais Solaris e Linux, embora ambos sejam baseados na plataforma Unix. Ele utilizou as facilidades oferecidas pelos sistemas operacionais para monitorar processos ao nível de usuário como a chamada de sistema *ptrace* e o sistema de arquivos `/proc`. Sistemas

como o Solaris 2.4 oferecem acesso ao monitoramento de processo através do sistema de arquivos virtual `/proc`. Esta interface permite um controle direto ao espaço de endereçamento do processo monitorado. Além disso, permite um controle de granulosidade fina, assim como provê um modo para o monitor abortar uma requisição de chamada de sistema antes que ela seja executada pelo processo monitorado [WAG 99]. O Solaris oferece uma forma mais fácil de monitoramento de processo para determinar os argumentos e retornar valores da chamada executada pelo processo monitorado, atendendo assim as necessidades de implementação do Janus. Por essas razões é que foi escolhido o Solaris 2.4.

Na implementação para Linux, foi usada a chamada de sistema *ptrace*, a única primitiva de monitoramento suportada pelo sistema [WAG 99]. A chamada de sistema *ptrace* apresenta algumas deficiências para as necessidades do Janus, como: possui uma granulosidade grossa, isto é, monitora todas as chamadas ou nenhuma. Outra limitação do *ptrace* é que várias implementações de sistemas operacionais não oferecem forma para que o processo monitor aborte uma chamada; isso tornou a chamada *ptrace* inadequada para o propósito. Porém, o Linux tem uma vantagem poderosa; o seu código fonte. Isso torna o Linux uma plataforma bem apropriada para experimentos de extensões de sistema operacional, levando assim o autor do Janus a estender a chamada *ptrace* com a finalidade de atender as necessidades de suporte do Janus. A implementação mais sofisticada da primitiva de monitoramento é chamada de *ptrace*++.

3.6 PBeam

O PBeam é um sistema que faz o balanceamento da carga para aplicações distribuídas em *clusters* de estação de trabalho [PET 98b]. Para tratar os processos, deve-se ter a localização e o tempo de execução do processo; se um processo é movido para um outro módulo ou a aplicação é reinicializada por um *checkpoint*, as referências para os objetos devem ser trocadas [PET 98a]. Porém, para prover transparência, a aplicação deve ser habilitada para trabalhar com as mesmas referências, desconsiderando a existência de movimento no espaço ou tempo.

No sistema PBeam é introduzido um esquema totalmente virtual de espaço de nome para os processos IDs, que transporta os endereços e nomes de arquivos. Tabelas de processos virtuais e de endereços virtuais são tratadas para mapear entre os nomes virtuais e reais. As chamadas de sistema das aplicações são capturadas, e os seus parâmetros são trocados para o virtual para dar suporte ao espaço de nomes corrente; então o serviço do sistema real é executado [PET 98a]. Da mesma forma, os valores de retorno são trocados para o real. Para evitar uma colisão de nomes, os processos são determinados globalmente com um único processo ID virtual.

O componente de interposição das chamadas de sistema do PBeam é separado do administrador do espaço de nomes, denominado de *pbeamd,* assim torna facilitada a troca entre as diferentes versões de ambos. Para explorar escalabilidade, também é implementada uma versão centralizada e outra distribuída do administrador de espaço de nomes [PET 98b]. O componente de interposição das chamadas de sistema tem três principais tarefas: (i) durante a operação normal, ele realiza a tradução do nome; (ii) quando é realizado um *checkpoint* ou migração, ele captura o estado interno e externo, salva para o disco, por TCP para um servidor de *checkpoint,* ou para outra máquina para

reconstrução imediata (ex. migração); (iii) ler um estado salvo para o disco ou para uma conexão de rede e reconstruir um processo corrente desta informação de estado. Outros componentes do sistema são compartilhados: o escalonador (*scheduler*), e um interceptador (*dispatcher*), que executa as decisões produzidas pelo escalonador.

A operação do processo de interposição de chamadas de sistema (SCIP) tem como meta prover a mesma transparência de *rollback* e migração que é alcançada quando um executável é ligado a um sistema de biblioteca modificada. Ou seja, possibilitar à aplicação que não pode ser ligada a estas bibliotecas modificadas possuir essa transparência [PET 98a, PET 98b]. Esse é um caso típico de programas comerciais. As interfaces de monitoramento dos sistemas operacionais oferecem serviços para acessar o estado interno de outros processos. O PBEAM explora esta forma transparente para executar as aplicações sobre o espaço de nome virtual. O SCIP usa a interface *ptrace*() para interpor e estender as chamadas de sistema, embora isto torne a implementação mais difícil e menos portável. No projeto do SCIP, foram considerados especificamente os sistemas SunOS, Solaris, Linux e Irix. Atualmente, a principal plataforma de implementação é um *cluster* de estações de trabalho executando SunOS 4.1, o qual não possui um sistema de arquivos virtuais `/proc`, mas diferentes extensões do *ptrace*().

A interposição das chamadas de sistemas consiste de diferentes fases:

- execução do processo até a próxima entrada de chamada de sistema.
- notificação de uma entrada de chamada de sistema. Por exemplo, recebe a notificação de que um processo deseja executar uma chamada de sistema. Ele deve ser parado antes da execução da chamada de sistema ser executada. Os argumentos da chamada, incluindo o número desta, devem ser lidos.
- trocar os argumentos, por exemplo, os nomes virtuais que devem ser procurados no espaço de nomes mapeado e trocados no espaço de nomes real.
- executar a chamada.
- notificação da saída da chamada e seus valores de retorno.
- mudar novamente o valor de retorno.
- reinicializar a última chamada de sistema, se necessário.

3.7 Phoinix

O Phoinix é um ambiente desenvolvido para tolerância a falhas, que é compatível com o *framework* OMA. No Phoinix, objetos de serviço podem ser desenvolvidos com capacidade de tolerar falhas, de *hardware* e *software* [DER 96]. A capacidade de tolerância a falhas no Phoinix é classificada sob três níveis: (i) re-inicialização, (ii) recuperação por retorno, e (iii) replicação. Atualmente o Phoinix, está

portado para Orbix 3.0 executando sob o SunOS 4.2. Os objetos de serviço providos na versão atual estão habilitados para tolerar falhas de *hardware* com a capacidade de recuperação por retorno.

O projeto do Phoinix tem como intenção desenvolver uma ambiente no qual a implementação de objetos é construída com capacidade de tolerância a falhas de uma forma semi-automática. A implementação de objetos com capacidade de tolerância a falhas de níveis 1, 2 e 3 é chamada *restart objects*, *logable objects*, e *replicated objects,* respectivamente. Como os nomes sugerem, os *restart objects* reiniciam o serviço após se recuperarem da falha, enquanto que os *logable objects* retornam o seu serviço para o seu último ponto de recuperação. Para os *replicated objects*, diferentes réplicas do objeto implementado coexistem no sistema de modo cooperativo. No estado atual, dois níveis de tolerância a falhas são suportados, isto é, o serviço de reinício (*restart service*) e o serviço de recuperação por retorno (*checkpoint-recovery service*).

Os principais componentes do Phoenix são os seguintes: compilador EIDL, a biblioteca de tolerância a falhas e o serviço de *log*.

O compilador IDL estendido (EIDL) troca a interface IDL do arquivo de especificação e produz códigos tolerantes a falhas em acréscimo ao código do compilador IDL padrão. Para o cliente, o compilador EIDL produz um *proxy* confiável para cada interface IDL. Este *proxy* é responsável pela detecção de falhas da implementação do objeto e por ativar o processo de recuperação de acordo com o tipo de implementação do objeto. Para a implementação do objeto, o compilador EIDL gera um *skeleton* tolerante a falhas. O *skeleton* executa a requisição de *log* nas operações normais e executa o *redo* destas requisições durante a recuperação de uma falha.

A biblioteca de tolerância a falhas consiste de duas famílias de classes para os *logable objects*. Uma família é para o *skeleton* tolerante a falhas, quando são definidas as classes de requisições persistentes (*Persist Request class)* e de tratadores de requisições *(Request Handler class),* classes-base para requisições de *logs* e de refazer (*redo*). A outra família define as classes-base para a aplicação herdar *logable objets*. Estas classes definem as funções virtuais *SaveStat()* e *LoadState()* para gerenciar os dados críticos e os membros do *logable objects*.

O servidor de *log* mantém registros de auditoria e também um armazenamento confiável para aquela implementação de objeto registrado.

3.8 AQuA

O AQuA (*Adaptive Quality of Service Availability*) [CUK 98, CUK 99] é uma arquitetura desenvolvida na Universidade de Illinois, cuja meta é permitir que aplicações distribuídas requeiram e obtenham um nível de disponibilidade desejada usando um *Quality Object* - QuO através de um administrador de propriedades. É composta de diferentes componentes como: *Quality Object*, *Proteus*, Maestro/Ensemble e *gateway* [CUK 99], conforme ilustrado na figura 3.7. O *framework* AQuA usa o QuO *runtime* para processar e fazer requisições de disponibilidade, o Proteus para gerenciar a confiabilidade e configurar o sistema em resposta as falhas e requisições de disponibilidade. Utiliza o Maestro/Ensemble para prover o serviço de comunicação de

grupo. Em adição, uma interface CORBA é provida para objetos da aplicação que usam *gateway* AQuA. O *gateway* traduz a comunicação entre os níveis do processo, como a suportada pelo Ensemble, e mensagens IIOP, compreendidas pelos ORBs em CORBA. O *gateway* também suporta uma variedade de tratadores, como os usados pelo Porteus para tolerar falhas de colapso (*crash)*, de valor e de tempo.

Antes de descrever a arquitetura AQuA, mais detalhadamente, é interessante fazer um breve comentário sobre as tecnologias que são usadas no AQuA para dar suporte à comunicação de grupo (Ensemble) e à especificação do serviço de qualidade (QuO).

Os grupos podem ser usados em sistemas de computação distribuída para ajudar administrar a complexidade de grandes aplicações ou para auxiliar a prover propriedades não funcionais, com confiabilidade e segurança. Com a finalidade de prover tolerância a falhas em um nível mais básico, o AQuA utiliza o sistema de comunicação de grupo Ensemble, para assegurar comunicação confiável entre grupos de processos e também para garantir atomicidade na entrega do *multicast* para grupos com mudança de membros, e para detectar e excluir do grupo os membros que falham por colapso (*crash)*. O protocolo do Ensemble usado no AQuA provê comunicação de grupo baseada no modelo de sincronismo virtual. Tanto o *multicast* com ordenamento total como causal são usados na estrutura de grupo do AQuA, resultando em um ordenamento total na entrega de mensagens entre diferentes grupos de réplicas de objetos. A arquitetura AQuA usa o mecanismo de detecção de defeitos do Ensemble e provê entradas para o Proteus ajudar na recuperação. O Maestro é utilizado por oferecer uma interface orientada a objetos em C++ para o Ensemble, possibilitando assim que aplicações orientadas a objetos possam ser escritas por derivação das classes do Maestro que oferecem comunicação confiável.

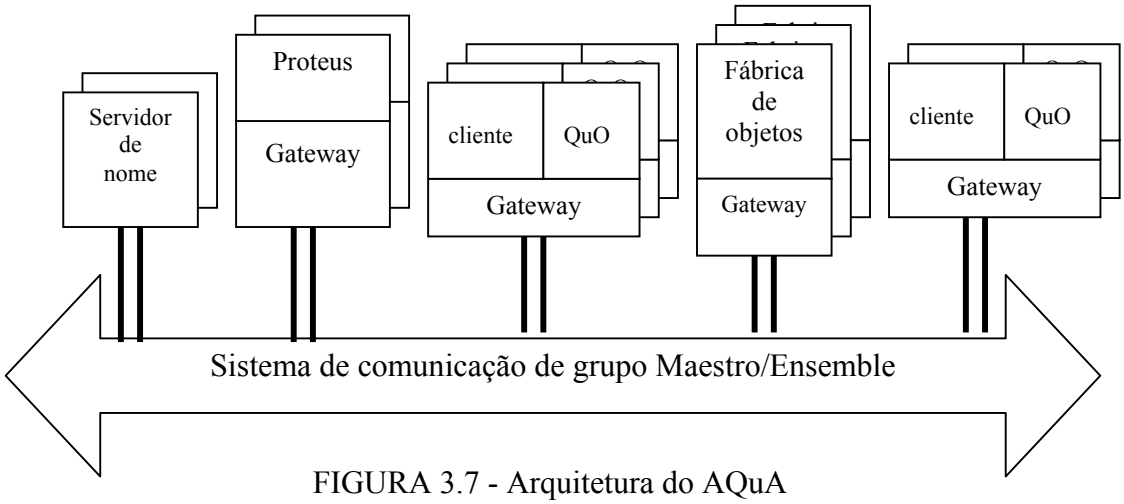


FIGURA 3.7 - Arquitetura do AQuA

O objetivo dos *Quality Objects* - QuO, é desenvolver um *framework* comum, baseado em computação de objetos distribuídos, que possa administra e integrar propriedades não funcionais, tais como restrições de recursos nas redes, requisições de confiabilidade e necessidade de segurança [CUK 98, SCH 99]. O QuO é utilizado no AQuA para transmitir requisições confiáveis da aplicação para o Proteus, o qual tenta configurar o sistema para ativar a disponibilidade desejável. O QuO também possui um mecanismo de adaptação que é usado quando o Proteus é incapaz de prover um nível específico de disponibilidade.

A arquitetura AQuA é planejada para dar suporte aos mais diferentes níveis de troca de condições que atingem a confiabilidade do sistema. O Proteus busca responsabilidade para configurar o sistema para assegurar confiabilidade e para responder a falhas individuais. Porém, têm casos onde a aplicação altera o requisito de disponibilidade, e o Proteus não é capaz de entregar o nível de requisição de disponibilidade. Neste caso, requer uma adaptação de cooperação entre a aplicação e o administrador de confiabilidade do Proteus. O QuO *runtime* provê serviços para facilitar esta cooperação, o mais importante desse serviço são os contratos, sistema de condições de objetos, e os *delegates*. Os contratos provêem um sumário de alto nível de níveis de requisições de confiabilidade feitos pela aplicação e o nível de confiabilidade entregue inicialmente. O contrato é acessado através do sistema de condições de objetos e delegados. O sistema de condições de objetos oferece uma interface muito simples para o programador de aplicações; tipicamente exporta somente métodos *set_value* e *get_value*. O QuO *delegate* é um componente do *software* que provê o comportamento para o *QoS-adaptive*, mostrando o comportamento funcional de um objeto *proxy* ORB

O Proteus [SAB 99] é uma estrutura flexível desenvolvida para adaptar tolerância a falhas na arquitetura AQuA. É composto pelo *dependability manager* que é replicado e consiste de duas partes, um *advisor* e um *protocol coordinator*:

- o *advisor* produz as decisões para reconfigurar o sistema, baseado nas informações de defeitos que ocorreram e as requisições de confiabilidade da aplicação via QuO *runtime;*

- o *protocol coordinator,* juntamente com o *gateway*, implementa a abordagem escolhida para tolerar a falha. Dependendo a opção tomada pelo *advisor*, o Proteus pode tolerar e recuperar falhas de colapso (*crash)*, falhas de temporização e falhas de valor em objetos da aplicação e do QuO *runtime*.

Os *Object factories* são usados para descontinuar ou iniciar aplicações replicadas, dependendo da decisão tomada pelo *dependabitity manager,* e para oferecer informações considerando o *host* para o *dependabitity manager*.

Os *gateways* oferecem duas funções: a primeira provê uma interface-padrão para aplicações. CORBA provê os desenvolvedores da aplicação com uma interface-padrão para construir aplicações orientadas a objetos distribuídas, mas não provê uma abordagem simples que permita que as aplicações sejam tolerantes a falhas [SAB 99]. O *gateway* oferece uma interface-padrão para traduzir a comunicação entre níveis de processo, como a suportada pelo sistema Ensemble, e mensagens IIOP, as quais são compreendidas pelos ORBs CORBA [SAB 99, SCH 99]. Dessa forma, aplicações distribuídas baseadas em CORBA escritas para a arquitetura AQuA podem usar o padrão dos ORBs disponíveis comercialmente.

A segunda função oferecida pelos *gateways* é a de prover tolerância a falhas usando vetores e protocolos de replicação [CUK 99]. Ambas as replicações (ativa e passiva) são suportadas pelos objetos AQuA. Esses serviços estão localizados nos *gateways handlers*.

O *gateway* AQuA é implementado como um processo, e é uma parte de um objeto AQuA. Ele intercepta mensagens IIOP geradas por um objeto CORBA e então transmite, usando o sistema de comunicação de grupo Maestro/Ensemble, para outro *gateway* [SAB 99]. O *gateway* é composto de um codificador/decodificador IIOP (IIOP *encode/decode)*, um interceptador (*dispatcher*), um tratador (*handler*), e uma interface para o Maestro/Ensemble [CUK 98, SAB 99].

O codificador/decodificador IIOP é usado como interface com um objeto CORBA [SAB 99]. Quando uma mensagem IIOP é capturada pelo decodificador, ele remove o cabeçalho IIOP e passa a mensagem para o interceptador, o qual a entrega na forma definida pelos protocolos do sistema. O *gateway* tem filas de mensagens provenientes e destinadas ao codificador/decodificador para serem usadas pelo interceptador, o qual separa o codificador/decodificador IIOP dos mecanismos do *gateway*, que tratam as mensagens uma vez entregues. O codificador lê as mensagens para a fila de entrada, empacota as informações sob uma mensagem IIOP, e a envia para a aplicação CORBA.

As interfaces do interceptador, como o codificador/decodificador IIOP, provêem um conjunto de características funcionais para entrega de mensagens. A função principal do interceptador é empacotar informações extras à mensagem IIOP, removida da fila de saída, e entregar a mensagem para o tratador correto, o qual se encarrega de receber e enviar mensagens. Essas mensagens empacotadas são chamadas de "*gateway messages*". O interceptador também o armazena o invólucro das requisições que ele recebe do tratador. Isto é feito de tal forma que o invólucro de uma requisição IIOP possa ser transferido para a resposta IIOP, quando a resposta é retirada da fila do decodificador. A "*gateway message*", entre outras informações, contém dados para esquemas de replicação e entrega correta de mensagens.

Os tratadores são responsáveis pelo envio e recebimento de mensagens. Um tratador é criado no *gateway* para cada par de objetos replicado que deseja comunicar-se. Três tipos de tratadores são suportados no *gateway*: tratador de réplicas ativas, tratador do administrador de confiabilidade e o tratador da fábrica. Diferentes tipos de tratadores são usados dependendo do tipo do grupo de objetos que está se comunicando. Cada *gateway* mantém um tratador de informações de estado para assegurar a correta entrega das mensagens.

3.9 GroupPac

O *GroupPac* segue a linha de soluções abertas definidas pela OMG, onde novas funcionalidades adicionadas nas especificações CORBA são definidas na forma de objeto de serviço comum, mantendo com isto o ORB inalterado [LUN 2000a]. A idéia básica é fornecer um conjunto de serviços e facilidades para construção de aplicações tolerantes a falhas. Dentro desta filosofia de objetos de serviço, o *GroupPac* oferece um conjunto de blocos de construção (*building blocks*). Esses objetos de serviço, que podem ser arranjados de diferentes formas no sentido de compor diferentes esquemas ou arquiteturas de serviço de aplicação com propriedades de tolerância a falhas, também podem ser combinados para dar suporte a aplicações não enfatizando necessariamente tolerância a falhas [LUN 2000a, LUN 2000b]. Aplicações distribuídas, tais como: *groupware*, ou mais precisamente aplicações de trabalho cooperativo, podem fazer uso

dos objetos desse pacote para implementar características ou facilitar aspectos de coordenação.

Os objetos que fazem parte da composição do GroupPac, SI, STE, SM, SDF, e SCG, são definidos logo a seguir. Cada objeto possui uma interface horizontal e uma vertical, ambas definidas em IDL/CORBA. A interface horizontal viabiliza a comunicação entre os objetos de serviço do mesmo tipo; já a interface vertical é o meio pelo qual é fornecido o serviço para os outros objetos [LUN 99a, LUN 2000b]. Esses objetos têm as seguintes finalidades:

- objeto SI (Serviço de Iniciação) - conforme o seu próprio nome sugere, ele é responsável pela iniciação de outros objetos de serviço do *GrouPac*. É a partir desse objeto que é construída a configuração necessária ao suporte de uma aplicação replicada.

- o objeto STE (Serviço de Transferência de Estado) - oferece funcionalidade para transferência de estados de um objeto para outro. Esse serviço é usado, por exemplo, em modelos de replicação ou para migração de objetos.

- objeto SM (Serviço de *Membership*) - é responsável pelo serviço de gestão de grupo de réplicas. Essa gestão deve ser transparente à aplicação, exercendo um controle dinâmico nas entradas e saídas de objetos de um grupo pela manutenção de listas atualizadas de seus membros.

- Objeto SDF (Serviço de Detecção de Falhas) - envolve um conjunto de procedimentos de detecção de falhas de objetos em um grupo. Esse objeto opera em conjunto com o objeto SM: quando o SDF detecta um *crash* de um dos objetos do grupo, essa falha é imediatamente reportada ao objeto SM para que esse gere uma nova lista de membros.

- O objeto SCG (Serviço de Comunicação de Grupo) - que oferece um conjunto de protocolos confiáveis de comunicação de grupo, com várias políticas de ordenação de mensagens (FIFO, Causal ou Total), construídos a partir de alguns objetos de serviço, como o SM, e de comunicações simples, ponto a ponto, ao nível do ORB.

Quanto aos aspectos de implementação do *framework*, todos os seus serviços são implementados na linguagem Java, que oferecem a vantagem da portabilidade, e suas interfaces especificadas segundo o padrão da IDL/OMG. A abordagem adotada pelo *GrouPac* é definida como uma abordagem híbrida [LUN 2000a], pois esta proposta combina características da abordagem de serviço e de interceptação, onde os aspectos de gestão de réplicas são providos por um conjunto de objetos de serviço em nível do ORB, segundo a abordagem de serviço. Aspectos, tais como a comunicação de grupo, seguem a abordagem de interceptação, onde conceitos como o de interceptador definido pela OMG são utilizados.

3.10 SLIC

O SLIC é um mecanismo de extensão que se utiliza da técnica de interposição para inserir extensões de código confiáveis no *kernel* de sistemas operacionais como Solaris e Linux. Tem como objetivo simplificar significativamente o processo de evolução e extensão dos sistemas operacionais por permitir uma larga classe de extensões e, em particular, as que possam administrar recursos globais ou reforçar as garantias de segurança [GHO 98]. A abordagem adotada na implementação do SLIC é a de adicionar novas funcionalidades ao sistema operacional sem modificações significativas. Isto é, adiciona novas funcionalidades com pequenas modificações no sistema operacional e não modifica o código executável (binário) das aplicações. Usa a interposição em nível de *kernel*, é simples de implementar, eficiente e não requer suporte de *hardware* [GHO 98]. O projeto e conseqüentemente a sua implementação buscam investigar a técnica de interposição para adicionar novas funcionalidades a sistemas operacionais do tipo Unix.

A arquitetura do SLIC é composta de múltiplos interceptadores (*dispatchers*) e extensões (*extensions*) com várias rotinas de suporte, conforme ilustrado na figura 3.8. Os interceptadores são responsáveis por capturar eventos na interface do sistema e passá-los para as extensões interessadas. A extensões recebem os eventos do interceptador e implementam novas funcionalidades. As rotinas de suporte providas pelas extensões são como uma simples interface de consistência para funcionalidades como alocação de memória e primitivas de sincronização. Essas rotinas permitem que a implementação do SLIC seja portável para vários sistemas operacionais [GHO 98]. Cada interceptador deve oferecer rotinas de suporte adicionais apropriadas para a interface; por exemplo, um interceptador de sinais deve prover rotinas para um determinado conjunto de sinais aguardados por um processo.

Cada interceptador é responsável por capturar eventos em uma única interface do sistema, como chamadas de sistema, sinais e interrupções de *hardware* [GHO 98]. O interceptador usa duas técnicas diferentes para capturar os eventos de interface:

- para as interfaces que usam tabelas de endereçamento, como as chamadas de sistema, e interfaces de memória virtual, o SLIC registra o endereço da função original na tabela de endereçamento e armazena o endereço de sua própria rotina de interceptação;

- em interfaces procedurais, as quais são chamadas diretamente de várias locais no *kernel*, como a entrega de sinais no Solaris, os interceptadores capturam os eventos usando um binário embutido. Primeiro, algumas poucas instruções de procedimentos relevantes são salvas e substituídas como instruções endereçadas pelo interceptador, sempre que o procedimento pode ser chamado. Quando a rotina original precisar ser invocada, as instruções salvas pelo procedimento de interceptação são executadas e o controle é retornado para a rotina original na instrução seguindo o binário embutido. Usando estas técnicas, o interceptador SLIC pode capturar invocações de interfaces pelo custo de uma chamada a procedimento.

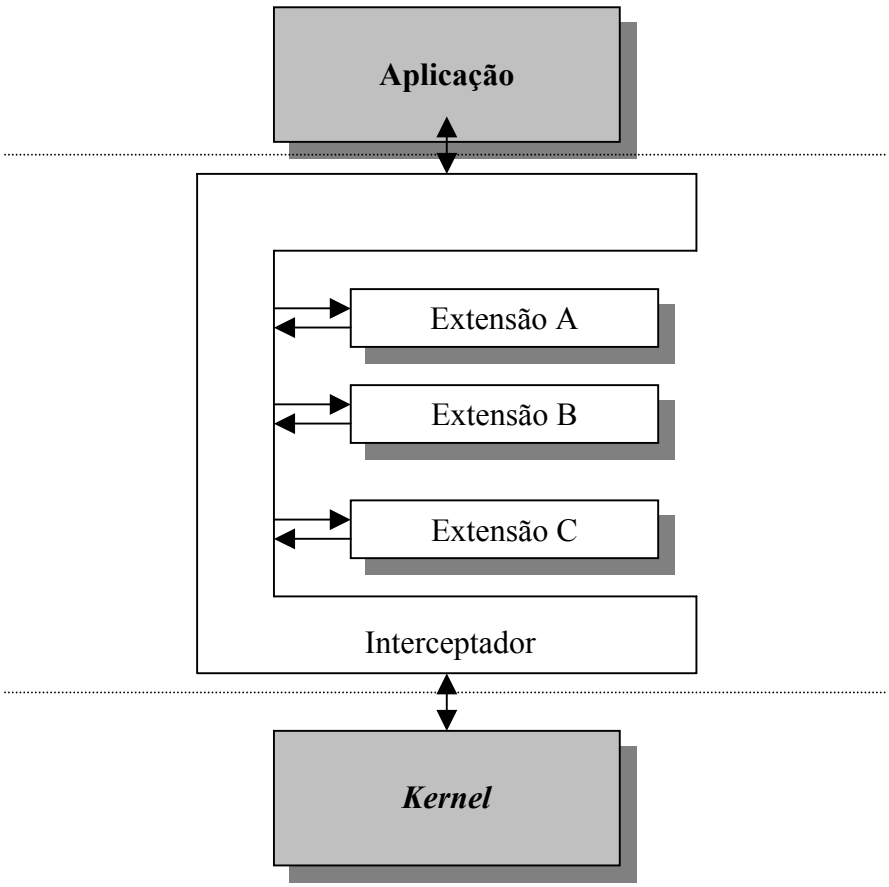


FIGURA 3.8 - Arquitetura básica do SLIC

A extensões podem ser carregadas em diferentes formas: como um serviço em nível de usuário (figura 3.9) ou como extensões no *kernel* [GHO 98] (figura 3.10) ou como uma combinação dos dois tipos.

- Executar extensões como um serviço permite um desenvolvimento para agir como uma aplicação de usuário, com acesso a bibliotecas disponíveis nesse nível. Uma desvantagem desta abordagem é o custo da troca de contexto.

- Uma extensão carregada diretamente dentro do *kernel* é invocada diretamente pelo interceptador por uma chamada a procedimento. Quando os eventos são freqüentes, esta organização possui melhor desempenho que a abordagem em nível de usuário.

- O uso simultâneo de ambas abordagens pode ser feito, onde uma parte pode ser alocada junto ao *kernel*, enquanto que as funcionalidades raramente usadas ou aquelas que requerem acesso a bibliotecas disponíveis em nível de usuário podem estar agregadas ao serviço nesse nível.

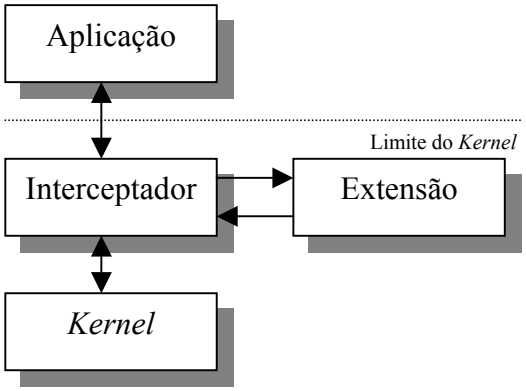


FIGURA 3.9 – Interface para Extensão adicionada em nível do *kernel*

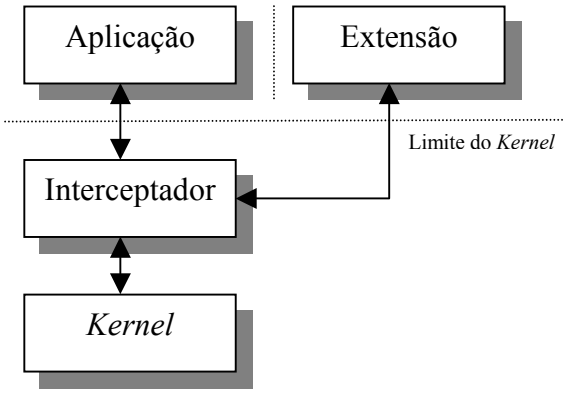


FIGURA 3.10 – Interface para Extensão em nível do usuário

3.11 *Kernel* Hypervisor

O chamado *Kernel Hypervisor* é implementado sobre o *kernel* do sistema operacional. Ele pode ser usado para fazer um componente mais robusto ou para executar várias funções de segurança, como controle de acesso e auditoria de eventos. Os *kernel hypervisors* são módulos carregados que capturam chamadas de sistema em seu pré- ou no pós-processamento delas [MIT 2000]. Ele também pode ser usado para prover uma camada adicional de controle de segurança de granulosidade fina ou para prover suporte à replicação. A meta do *Kernel* Hypervisor é a proteção contra códigos maliciosos ou para prover algum tipo adicional de funcionalidade.

A composição da arquitetura do *Kernel Hypervisor* [MIT 2000] é formada de três principais componentes:

- o *kernel hypervisor* mestre - administra individualmente os *kernel hypervisors*-cliente que são carregados, e provê uma facilidade de controle permitindo que o usuário monitore e configure esse *hypervisor*-cliente.

- *hypervisor* de *kernel* de cliente específico - provê a aplicação para fazer uma política de decisões específica. Cada *hypervisor kernel* de cliente pode cobrir uma ou mais aplicações. O *user daemons* que executa no espaço do usuário também pode ser desenvolvido para permitir que o *hypervisor* cliente inicie ações no espaço do usuário.

- módulo de administração do *hypervisor* cliente - provê uma interface para comunicação e configuração do *hypervisor kernel* de cliente para o espaço do usuário.

A utilização do *Kernel Hypervisor* oferece várias vantagens [MIT 2000], como: ele não requer modificações do *kernel*, todo o código do *Kernel Hypervisor* é implementado em um módulo e carregado dentro do *kernel* enquanto o sistema é executado. O *Kernel Hypervisor* é flexível, pode ser usado tanto para implementar variedade de diferentes tipos de política de segurança e para prover funcionalidade de replicação. Ele pode ser usado em qualquer sistema operacional que suporte a carga de módulos que possuam acesso à estrutura de dados das chamadas de sistema, nestes sistemas inclui-se o Linux, Solaris e outros sistemas modernos baseados em Unix.

Utilizando o conceito de módulo, a implementação do *Kernel Hypervisor* foi realizada em um *kernel* Linux. A escolha do Linux como uma plataforma inicial deve-se ao fato que ele suporta a carga de módulos, possui o código-fonte livre e facilita que a avaliação dos resultados do trabalho torne-se acessível a outros pesquisadores e desenvolvedores [MIT 2000].

4 Ambiente de implementação

Após um levantamento bibliográfico objetivando a busca principalmente de aspectos referentes à implementação das abordagens conceituadas no capítulo 2, identificou-se que estas podem ser implementadas em nível de usuário ou de *kernel*. Diante dessa e de outras informações, a tarefa seguinte a ser realizada foi a identificação do ambiente de suporte, isto é, o ambiente que ofereceria as condições necessárias para a implementação das abordagens. Julga-se que seria interessante, do ponto de vista de comparação, que houvesse, tanto quanto possível, uma certa uniformidade quanto ao tipo de programação empregado na implementação dos mecanismos, para evitar privilegiar de antemão algum dos enfoques, do ponto de vista comparativo. O primeiro nível tomado por opção foi o das linguagens de programação, que costuma ser mais amigável que os níveis inferiores. Entretanto, a idéia inicial de definir os mecanismos diretamente sobre uma linguagem de programação, como as versões-padrão de C ou C++, não se mostrou viável. Essas linguagens não oferecem mecanismos capazes de suportar as características essenciais de implementação da abordagem de interceptação como, por exemplo, a captura de mensagens.

Assim, a investigação voltou-se para mecanismos disponíveis em sistemas operacionais. A opção pela utilização do Linux visou atender alguns requisitos importantes para o desenvolvimento dos protótipos tais como: facilidade de instalação, boa documentação, código aberto. Este último foi considerado um ponto essencial para o desenvolvimento desse trabalho, devido ao fato que a construção dos protótipos[5] exige uma programação em diferentes níveis: dentro do sistema, em camada entre o sistema e a aplicação e como um processo independente. Outro fator que veio reforçar a opção pelo Linux foi a localização de alguns trabalhos que fazem uso dos mecanismos similares aos buscados neste trabalho, oferecidos pelo sistema operacional, como: Ufo [ALE 99], Janus [WAG 99], SLIC [GHO 98], descritos no capítulo 3. Embora, eles não compartilhem da mesma nomenclatura adotada na classificação de Felber [FEL 98], eles utilizam essas abordagens, buscando oferecer as mais diversas funcionalidades. Um outro requisito importante do ponto de vista da pesquisa é o livre acesso à plataforma Linux facilitando a que outros pesquisadores e desenvolvedores beneficiem-se dos resultados obtidos nesse trabalho.

Ao longo dessa pesquisa, percebeu-se que as suposições iniciais formuladas sobre os requisitos citados anteriormente nem sempre atenderam as expectativas de forma plena. Por exemplo, a documentação do sistema Linux é bastante satisfatória no que se refere à instalação e administração, mas quando se trata de alguns recursos disponibilizados pelo sistema, como as interfaces de monitoramento, a documentação deixa muito a desejar, apresentando-se pouco esclarecedora e, em alguns casos confusa, exigindo a busca de respostas através do estudo de seu código fonte. Mas, a sua complexidade dificulta esta atividade.

A escolha da linguagem de programação a ser utilizada teve como base o fato de que as implementações das diferentes abordagens precisam explorar tanto o nível do

[5] A implementação dos protótipos foi necessária para aprofundar a análise sobre as diferentes abordagens. Obtendo informações que foram classificadas neste trabalho como: quantitativas e qualitativas. Formando assim um perfil das abordagens de integração, interceptação e serviço.

kernel como o nível do usuário. Diante deste requisito, optou-se pela linguagem C que apresenta as características necessárias para dar suporte a ambos os níveis e, portanto, possibilitou a implementação homogênea dos protótipos com a mesma linguagem para a programação, permitindo uma comparação mais justa. Adicionalmente, outro aspecto motivador foi a compatibilidade total com o sistema operacional Linux.

4.1 Sistema operacional Linux

O Linux é uma re-implementação completa da especificação POSIX[6] para sistemas Unix, com extensões *System* V e BSD [MAX 2000, BER 98]. Embora seja um sistema operacional totalmente comparável com as implementações comerciais do POSIX, como SCO, Xenix, SunOS, apresenta um diferencial importante: sendo uma re-implementação completa, não usou nenhuma parte de código comercial existente. Possibilita assim a sua distribuição gratuita tanto na forma binário como em código fonte. Inicialmente foi desenvolvido para a arquitetura Intel(x86), mas atualmente é portável também em arquiteturas como: Alpha, Sparc, PowerPC, entre outras. O sistema é de distribuição livre, desde que observados os termos do GPL (*General Public License*) do GNU.

Todas as características que acompanham as várias implementações comerciais do Unix estão presentes no Linux onde, entre elas, estão os suportes a: multitarefas, multiusuário, multiplataforma, multiprocessamento, *multithreading* [BER 98]. O Linux em si é somente o *kernel* do sistema operacional [MAX 2000], a parte que controla o *hardware*, manuseia arquivos e organiza processos. Existem muitas "combinações" de *kernel* com pacotes de utilitários e aplicações, normalmente bastante diferentes, que são chamados de distribuições. A maioria dos utilitários e aplicações é genérica para Unix, de domínio público, e normalmente distribuída sob a GPL do GNU.

O Linux tem a sua estrutura composta de três camadas: utilitários e aplicativos externos, *shell* e *kernel*. O *kernel* controla dispositivos de *hardware*, administra recursos de processos disponíveis em relação aos processos dos usuários, planeja e executa tarefas internas e administra o armazenamento de dados, decide sobre o estado atual dos processos em execução, mantém arquivos e diretórios temporários ou de configuração, também estabelecendo e controlando suas permissões [MAX 2000]. O *shell* é a interface de utilização do Linux. Ele fornece um ambiente onde o usuário pode elaborar comandos e definir parâmetros.

4.1.1 Características do código fonte do Linux

Entre algumas das características do código-fonte do Linux está a utilização das linguagens C e Assembler em sua implementação [MAX 2000, BER 98]. Os prós e os contras usuais destas linguagens incluem: o código em C é mais fácil de manter, embora o Assembler seja mais rápido [MAX 2000]. O Assembler geralmente é usado quando a velocidade de execução é um ponto crucial ou quando algum recurso específico de

[6]O POSIX é uma especificação, um padrão que define como deve ser a interface de *software* entre as aplicações e o sistema operacional. A concordância como o padrão POSIX significa que o código desenvolvido em Linux poderá ser facilmente re-compilado em outros Sistemas Unix.

alguma plataforma exige, como é o caso do acesso direto ao *hardware* de gerência de memória. Quando isto ocorre, parte do *kernel* é compilada com o g++ (o compilador C++ da GNU), embora não faça uso dos recursos para objetos do C++.

O *kernel* do Linux é projetado para ser compilado com o compilador C do GNU [MAX 2000], o gcc. Um compilador C qualquer não servirá, pois o código do *kernel* em alguns casos utiliza-se de recursos específicos do gcc. Alguns dos códigos específicos do gcc simplesmente usam extensões da linguagem, como a palavra chave *inline* em C [MAX 2000] para designar uma função alinhada, isto é, cujo código possa ser expandido sempre que a função for chamada, poupando assim a necessidade de uma chamada real à função.

4.1.2 Divisão do *Kernel*

Nos sistemas baseados em Unix, vários processos são empregados na realização de tarefas diferentes, sendo que cada um deles exige recursos do sistema, seja na capacidade de computação, memória, conexão de rede ou outros recursos. O *kernel* é responsável pela maior parte do código executável que atende a estas exigências [RUB 99]. A função do *kernel* pode ser dividida nas atividades descritas a seguir, e representadas esquematicamente na figura 4.1.

Gerência de processos: o *kernel* é responsável pela criação e extinção dos processos e por estabelecer a conexão deles com o mundo exterior (entrada e saída). A comunicação entre processos, fundamental para o funcionamento global do sistema, também é uma tarefa do *kernel*, além da distribuição das tarefas. De um modo geral, a atividade de gerência de processos, realizada pelo *kernel*, implementa a abstração de vários processos sobre uma única CPU.

Na **gerência de memória,** o *kernel* aloca espaços de endereçamento virtual para todo e qualquer processo que esteja usando seus recursos disponíveis. As diferentes partes do *kernel* interagem com o subsistema de gerência de memória, utilizando para isto um conjunto de chamadas de funções, que variam desde recursos simples, equivalentes ao *malloc* e *free*, até funcionalidades mais complexas.

O Unix é um sistema fortemente baseado em um conceito de **sistema de arquivos**; deste modo, quase tudo no Unix pode ser tratado como arquivo. O *kernel* constrói um sistema de arquivos sobre um *software* sem estrutura e a abstração resultante é constantemente usada por todo o sistema. Além disso, o Linux oferece suporte a vários tipos de sistemas de arquivos, ou seja, formas diferentes de organizar os dados no meio físico.

Grande parte das operações do sistema mapeia um dispositivo, portanto utiliza-se do **controle de dispositivos**. Com exceção do processador, memória e alguns outros poucos componentes, todas e quaisquer operações de controle de dispositivo são executadas por um código que é específico para o dispositivo em questão. Esse código, chamados de *driver* de dispositivo, deve estar presente para cada periférico de seu sistema.

Por fim, a **gerência da rede** deve ser realizada pelo sistema operacional, pois grande parte de suas operações não é específica para um processo. Pacotes de entrada são eventos assíncronos. Eles devem ser coletados, identificados e despachados antes que um processo cuide deles. O sistema é responsável pela passagem dos pacotes de dados pelo programas e interfaces de rede, e deve desativar e ativar corretamente os programas que esperam pelos dados da rede. Além disso, todas as questões referentes à definição de endereços e roteamento são executadas dentro do *kernel*.

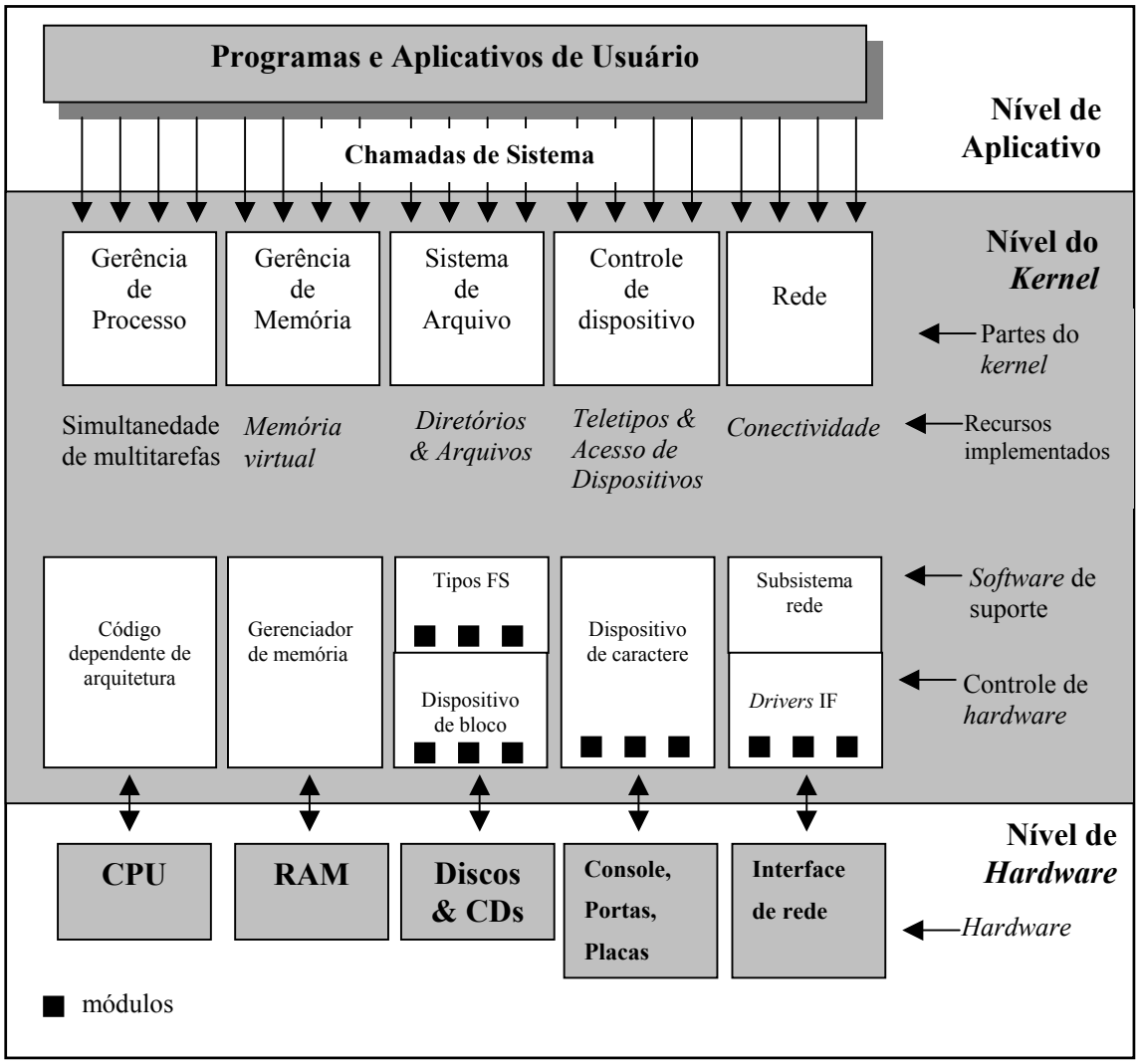


FIGURA 4.1 - Uma visão da divisão do *kernel*

4.1.3 Módulo & Aplicativos

Uma das características do Linux é a capacidade de expandir o código do *kernel* durante a execução. Isto significa que é possível adicionar novos recursos ao *kernel* sem haver a necessidade de reinicializar o sistema [BER 98, RUB 99, MAX 2000].

Cada parte do código que pode ser acrescentado ao *kernel* chama-se módulo. O *kernel* do Linux oferece suporte para alguns tipos ou classes bastante diferentes de módulos incluídos, mas não se limitando aos *drivers* de dispositivos (*device drivers*). Cada módulo é formado por um código objeto [BER 98], que se caracteriza por ser dinamicamente ligado ao *kernel* em execução, através dos programas *ismod* e desligado pelo programa *rmmod* [RUB 99]. É importante observar que somente o usuário com direito de execução em nível de "*root"* pode realizar estas tarefas.

Após esta breve contextualização, também é importante enfatizar algumas diferenças entre um módulo do *kernel* e um aplicativo, em sua maior parte extraída de [RUB 99].

Enquanto um aplicativo executa uma tarefa simples do início ao fim, um módulo faz seu próprio registro para atender a requisições futuras, conforme ilustrado na figura 4.2. A função *init_module()*, que é ponto de entrada do módulo, faz o registro do módulo junto ao *kernel*, possibilitando ao módulo atender solicitações posteriores. O segundo ponto de entrada, a função *cleanup_module*, é chamada imediatamente antes do módulo ser descarregado, informando ao *kernel* que ele não está mais disponível. Esta característica ajuda a diminuir o tempo de desenvolvimento, possibilitando a realização de testes de novos *drivers* sem a necessidade de passar pelo extenso ciclo de inicialização de desliga/reinicializa toda vez.

Uma outra diferença é que, quando se está programando um aplicativo, pode-se chamar funções não definidas por ele, utilizando-se dos recursos das bibliotecas de funções. Por outro lado, um módulo está ligado apenas ao *kernel* e as únicas funções que ele pode chamar são as oferecidas pelo *kernel.* Assim, a programação pode tornar-se mais trabalhosa, dependendo do tipo de recurso que se pretende oferecer através da implementação do módulo.

Já que não há uma biblioteca ligada aos módulos, no código-fonte destes, não devem ser incluídos os arquivos de cabeçalhos, normalmente encontrados nos aplicativos. Tudo que se refere ao *kernel* está mencionado nos cabeçalhos encontrados em */usr/include/linux* e */usr/include/asm*. Os arquivos de cabeçalho encontrados nestes diretórios também são usados indiretamente na compilação dos aplicativos. Em *kernels* recentes, também são encontrados os diretórios de cabeçalhos *net* e *scsi*, mas não é comum que os módulos utilizem-se deles.

Uma diferença importante entre a programação do *kernel* e dos programas aplicativos está em como lidar com as falhas, visto que uma falha de segmentação é inofensiva durante o desenvolvimento de um aplicativo e um depurador sempre pode ser usado para rastrear o erro no código-fonte. Uma falha no *kernel* é fatal para o processo atual, e possivelmente para todo o sistema.

Para resumir, pode-se tomar por base um conceito que faz parte dos fundamentos da teoria de sistemas operacionais: um módulo é executado no assim chamado espaço do *kernel*, enquanto que os aplicativos o são no espaço do usuário. No Unix, o *kernel* é executado no nível mais amplo, também chamado de "modo supervisor", onde tudo é permitido, enquanto os aplicativos são executados no nível mais restrito, também conhecido como "módulo do usuário", onde o processador inibiu

algumas ações como, por exemplo, acesso direto ao *hardware* e a memória. Enfim, referências aos modos de execução como **espaço do *kernel*** e **espaço do usuário**, correspondem a referências aos diferentes mapeamentos de memória e, em conseqüência, aos diferentes espaços de endereçamento usados pelo código do programa [MAX 2000].

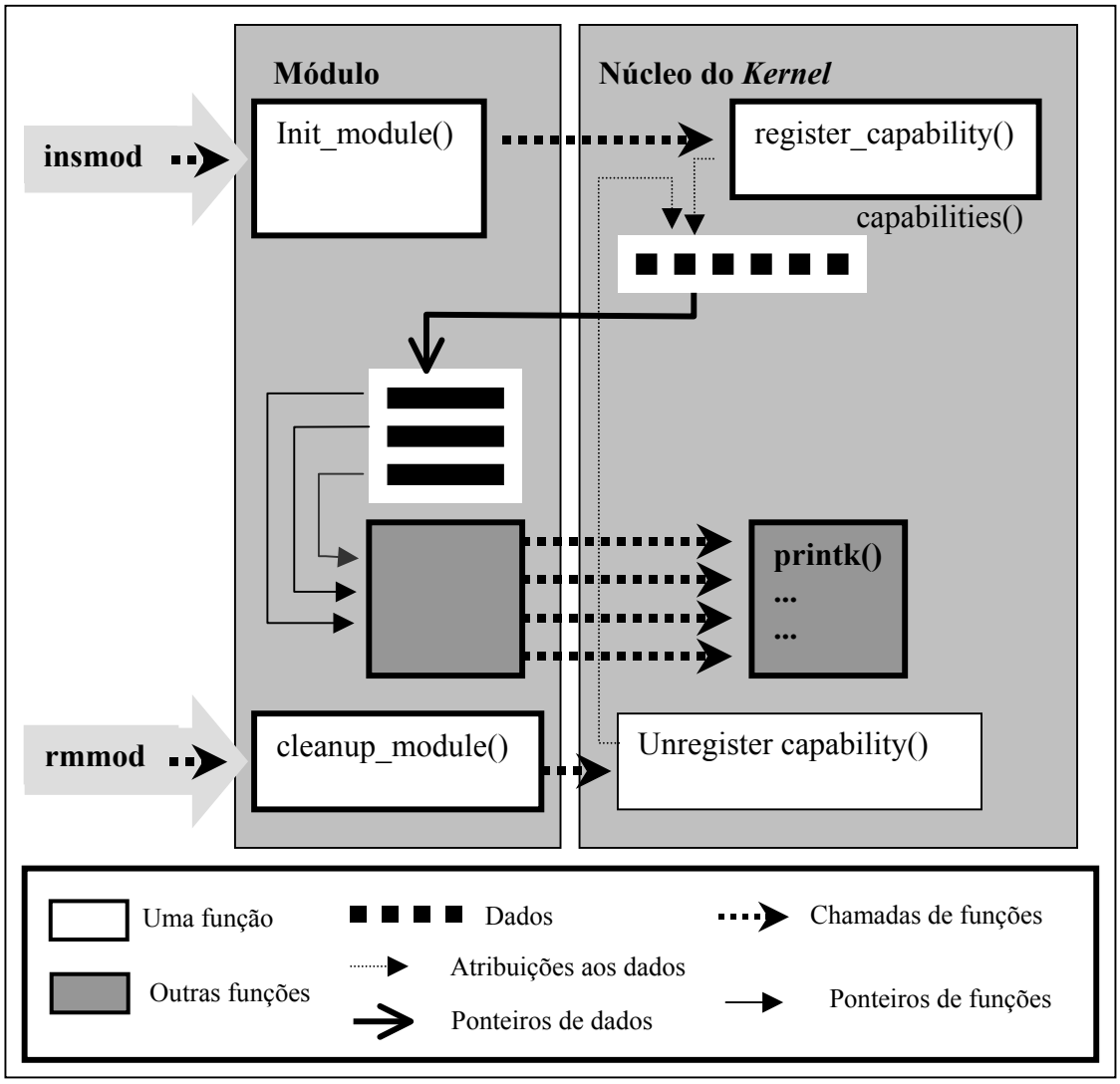


FIGURA 4.2 - Ligando um módulo ao *kernel*

4.1.4 Chamadas do sistema

As informações constantes nesta seção foram extraídas do livro de Maxwell [MAX 2000].

Uma chamada do sistema ocorre quando um processo de usuário solicita um serviço proporcionado pelo *kernel*. As chamadas guardam o acesso aos recursos que o *kernel* gerencia. As categorias maiores de chamadas do sistema são as que lidam com

E/S (*open, close, read* e outras), processos (*fork e execve,* entre outras), tempo (t*ime, settimeofday*, por exemplo) e memória (*mmap,bkr* e assim por diante); quase que a totalidade das chamadas encaixam-se em uma dessas categorias.

O valor de retorno de uma chamada do sistema deve ser sempre *int.* Por convenção, o valor retornado é zero ou positivo para indicar sucesso, retorno de um valor negativo tradicionalmente indica um erro (mas isto não é mais totalmente verdadeiro, será visto mais adiante). No caso de falha, as chamadas do sistema simplesmente retornam o valor negativo de um número de erro e a implementação da biblioteca C padrão fica responsável pelo resto. As funções de sistema do *kernel* em geral não são invocadas diretamente a partir do código do usuário, mas por um nível intermediário do código da biblioteca C padrão, que é responsável por este transporte.

Nos *kernels* mais recentes, nem todo valor negativo retornado por uma chamada representa um erro. Algumas poucas chamadas do sistema como a *lseek*, retornam valores negativos grandes mesmo no caso de sucesso. Atualmente os valores de erro estão em um intervalo entre -1 e -4.095, sendo que agora a biblioteca C padrão é responsável pela interpretação dos valores negativos retornados de uma forma mais sofisticada.

Para prosseguir na discussão sobre chamada do sistema, mais propriamente sobre como elas são invocadas, alguns termos devem ser previamente comentados como: interrupção, espaço do usuário e espaço do *kernel*.

O termo **interrupção** tem dois sentidos: interrupções de *hardware*, que não serão tratadas aqui por não serem relevantes para este contexto, e interrupções de *software*. Em uma CPU x86, estas interrupções são a forma de um processo do usuário sinalizar ao *kernel* que deseja fazer uma chamada do sistema. A interrupção usada para este propósito é a de número 0x80, mais conhecida pelos *hackers* da Intel como INT 80h. O *kernel* responde a esta interrupção com a função *system_call*, a qual será tratada logo a seguir.

Complementando o que foi abordado sobre os termos **espaço do *kernel*** e **espaço do usuário** na seção 4.1.3, estes fazem referencia à memória reservada para o *kernel* e a memória reservada para os processos do usuário, respectivamente. O espaço do usuário é onde diversos processos de usuários estão simultaneamente em execução e normalmente não compartilham memória.

A **invocação das chamadas do sistema** pode ser realizada de duas formas diferentes: a função *system_call* e a porta (*gate*) de chamada *lcall7*[7]. Neste trabalho, será abordada de forma mais detalhada somente a primeira forma. Como é possível perceber, as chamadas do sistema não são chamadas diretamente. Esta separação das chamadas dos mecanismos que as tornam acessíveis é bastante elegante. Se, por alguma razão, for necessário adicionar outro método para invocá-las, não é necessário alterá-las para suportá-lo.

[7] A função *syscall* é implementada como uma chamada para lcall7 [MAX 2000].

A função *system_call* é o ponto de entrada para todas as chamadas, isto é, para o código nativo, *lcall7* é utilizada para o suporte ao padrão iBCS[8] versão 2. A *system_call* é invocada pela biblioteca C padrão, que carrega os registradores da CPU com os argumentos que deseja passar, e, em seguida, deflagra a interrupção de *software* 0x80. O *kernel* registra a associação entre a interrupção de *software* a função *system_call*[9].

O primeiro argumento da *system_call* é o número da chamada do sistema a ser invocada. Ela também permite até outros quatro argumentos que serão passados adiante para a chamada do sistema. Nos raros casos em que o limite de quatro argumentos é oneroso, é possível contornar o problema tornando um dos argumentos um ponteiro para uma estrutura, que poderá então conter tantas informações adicionais quantas forem necessárias.

Enfim, a função *system_call* tem como tarefa conferir a validade da chamada e tomar diferentes caminhos dependendo dos resultados obtidos: caso seja uma chamada válida, informa ao *kernel* qual serviço está sendo solicitado pelo processo, além de verificar na tabela de chamadas, a *sys_call_table*, o endereço da função do *kernel* a ser chamada.

4.1.5 Chamada do sistema *ptrace*

Não é possível supor que a escrita de programas, em sua primeira tentativa, resulte em programas totalmente livres de erros. Todo o código escrito dever ser testado. Os sistemas *Unix* possuem uma chamada do sistema que provê a possibilidade de que um processo obtenha o controle sobre outro processo: esta chama denomina-se *ptrace*. O processo sob controle pode ser executado passo a passo e a sua memória pode ser lida e alterada [BER 98].

Depuradores como o *gdb, strace e ltrace* entre várias outras ferramentas baseiam-se nessa chamada. Por ser uma chamada de sistema e com isso ser parte integrante do *kernel* do sistema operacional e, por causa da dependência das características do processo, o código dessa chamada encontra-se definido no arquivo *arch/i386/kernel/ptrace.c*. A chamada de sistema *ptrace* no sistema operacional Linux possui a seguinte disposição [BER 98]:

```
int ptrace(long request, long pid, long addr, long data)
```

Esta função processa vários parâmetros definidos junto ao parâmetro *request*, os quais são descritos a seguir. No parâmetro `pid`, especifica-se a identificação do processo a ser depurado.

- Usando-se no parâmetro *request* a requisição *`PTRACE`*`_TRACEME`, um processo pode especificar que seu processo-pai o controlará via chamada do sistema *ptrace*, ou seja, o *flag* (`PF_TRACED`) de depuração do processo é ativado.

[8] A especificação iBCS2 se baseia em um *kernel* de interface padrão para aplicativos baseados em sistema Unix x86 que inclui, não apenas o Linux, mas também outros sistemas Unix x86 gratuitos como o FreeBSD, assim como o Solaris/x86, o SCO Unix, entre outros.
[9] A função *system_call* é, portanto, um tratador de interrupções.

- O processo-depurador pode fazer uso da requisição `PTRACE_ATTACH` para tornar qualquer processo seu processo-filho e ativar o *flag* de depuração. Porém, as identificações de grupo e usuário devem ser iguais às informações do processo depurado. O novo processo-filho recebe um sinal `SIGSTOP`, que irá interromper a sua execução. Após essa requisição, o processo estará sob controle de um novo processo-pai.

- Com exceção da requisição `PTRACE_KILL`, as descritas nos próximos itens são processadas pela função *ptrace* somente quando o processo-filho, o processo a ser depurado, está parado.

- As requisições `PTRACE_PEEKTEXT` e `PTRACE_PEEKDATA` podem ser usadas para ler palavras de 32 bits da área de memória do processo-filho. A requisição `PTRACE_PEEKTEXT` lê código, enquanto a requisição `PTRACE_PEEKDATA` pode ser usada para ler dados.

- A requisição `PTRACE_PEEKUSER` lê um valor do tipo *long* (32 bits), da estrutura *user* do processo que está sendo depurado. A *user struct* é uma estrutura virtual, onde as informações de depuração, como registradores de depuração do processo e o endereço de início da área de código são armazenadas. Estas informações são atualizadas pelo processador após uma interrupção de depuração e escrita na tabela do processo pela rotina de tratamento apropriada. A função *sys_ptrace()* usa o endereço a ser lido (*addr*), para definir que informação deve ser retornada e a providencia. Deste modo, os registradores na pilha do processo e os registradores de depuração armazenados na tabela de processos serão lidos com essa função.

- As requisições `PTRACE_POKETEXT` e `PTRACE_POKEDATA` possibilitam que a área de usuário do processo sob controle seja alterada, áreas de dado e código. Caso a área a ser modificada seja protegida contra escrita, a página de memória relevante é salva pelo processo de *copy-on-write*. Isso é usado, por exemplo, para escrever uma instrução em um ponto determinado do código de máquina para gerar uma interrupção de depuração. O código será executado até que a instrução de disparo da interrupção (*int3* no caso dos processadores X86) seja processada, ponto em que a rotina de gerência de interrupção de depuração interromperá o processo e modificará o seu processo-filho.

- Também é possível o uso das requisições `PTRACE_PEEKUSR` e `PTRACE_POKEUSR` para modificar a estrutura *user*. O principal uso dessas requisições é a modificação dos valores dos registradores do processo que está sendo depurado. Elas permitem que apenas o conteúdo de um registrador possa ser alterado a cada iteração. Com a mesma finalidade de trabalhar com os valores dos registradores existem as requisições `PTRACE_GETREGS` e `PTRACE_SETREGS`, possibilitando a leitura e alteração dos valores dos registradores do processo, sendo que a grande vantagem delas é que permitam que com apenas uma chamada *ptrace* seja possível ler ou alterar o conteúdo de todos os registradores ao contrário das

anteriores, oferecendo uma grande diferença de custo computacional. Com o uso das requisições anteriores, seriam necessários seis chaveamentos de contexto para a leitura de um valor na memória ou de um registrador.

- Após ser interrompido por um sinal, geralmente `SIGSTOP`, o processo-filho pode ser continuado pela requisição `PTRACE_CONT`. O argumento *data* pode ser usado para definir que sinal o processo irá tratar quando retornar à sua execução. Ao receber o sinal, o processo-filho informa ao processo-pai, que é o processo-depurador e pára. O processo-pai, então, pode continuar a execução do processo-filho e decidir se deve ou não tratar o sinal. Se o argumento *data* for nulo, o processo-filho não processará nenhum sinal.

- A requisição `PTRACE_SYSCALL` faz o processo continuar, da mesma forma que `PTRACE_CONT`, mas somente até a próxima chamada. A função *sys_ptrace()* então liga o *flag* `PF_TRACESYS` do processo. Quando o processo-filho chegar à próxima chamada do sistema, pára e recebe um sinal `SIGSTRAP`. O processo-depurador poderia neste ponto, por exemplo, inspecionar os argumentos da chamada do sistema. Se for dada continuidade ao processo com uma posterior requisição `PTRACE_SYSCALL`, o processo vai parar ao completar a chamada do sistema; o resultado e a eventual variável de erro podem ser lidas pelo processo-pai.

- A requisição `PTRACE_SINGLESTEP` diferencia-se da `PTRACE_CONT` porque liga o *flag* de interrupção do processador. O processo assim executa apenas uma instrução de código de máquina e gera uma interrupção de *debug* (nº 1). Isto liga um sinal `SIGSTRAP` para o processo que é interrompido novamente. Em outras palavras, `PTRACE_SINGLESTEP` permite que se execute o código de máquina, instrução por instrução. A requisição `PTRACE_KILL` continua o processo-filho enviando um sinal `SIGKILL`. O processo ao receber o sinal é abortado.

- A requisição `PTRACE_DETECH` separa o processo que está sendo controlado do processo-controlador, sendo que o primeiro é devolvido ao seu antigo processo-pai; seus *flags* `PF_TRACED` e `PF_TRACESYS` são desligados, juntamente com os *flags* de interrupção do processador.

A utilização da chamada *ptrace* por um depurador é feita da seguinte forma: o depurador executa a chamada do sistema *fork* e chama no processo-filho o *ptrace* com a requisição `PTRACE_TRACEME`. Então, o programa a ser inspecionado é iniciado via *execve*. Como o *flag* `PF_TRACED` está ligada, a chamada do sistema *execve* envia um sinal `SIGTRAP` para si mesmo. Existem, neste ponto, algumas verificações de segurança que não permitem à função *ptrace* trabalhar com programas que possuem o bit S ligado. A função deste bit é transferir ao usuário que executa o programa os direitos de acesso do proprietário ou do grupo proprietário do programa. Não fica difícil de imaginar as possibilidades que seriam abertas aos *hackers* se a função *ptrace* apresentasse um comportamento diferente.

No retorno da chamada *execve*, o sinal `SIGTRAP` é processado. O processo é parado e é enviado um sinal `SIGCHLD` ao processo-pai. O processo-depurador, o pai, aguardará por isso via chamada do sistema *wait*. A partir desse momento, o processo-pai pode inspecionar a área de memória do processo-filho, alterá-la e definir pontos de parada. A forma mais simples de se definir um ponto de parada em processadores x86 consiste em escrever uma instrução *int3* no endereço apropriado no código de máquina.

Se o depurador chama a função *ptrace* com a requisição `PTRACE_CONT`, o processo-filho continua a executar até encontrar uma instrução *int3*, ponto em que a rotina de interrupção apropriada envia um sinal `SIGSTRAP` para o processo-filho, o processo-filho é interrompido e o processo-depurador é novamente informado. O depurador poderá, então, simplesmente abortar o processo sob inspeção.

4.2 Característica do hardware e da rede empregados

A estrutura de suporte para a realização dos experimentos, com os protótipos das abordagens sob as aplicações Ping-Pong e Escolha de Líderes, foi constituído de cinco microcomputadores com a seguinte configuração: Pentium MMX, CPU 200MHz e 64Mb de memória RAM. Esses microcomputadores utilizavam como sistema operacional, o sistema Linux com *kernel* versão 2.2. A interligação entre esses computadores foi através de uma rede Ethernet de 10Mbit/s.

4.3 Aplicações escolhidas

Buscando satisfazer da melhor forma possível os objetivos a que este trabalho se propôs, ou seja, analisar os aspectos qualitativos e quantitativos das abordagens de captura de mensagens (interceptação, serviço e integração), foram escolhidas aplicações para a realização de experimentos práticos. Embora a principal motivação do trabalho fosse a de vincular a obtenção de informações da aplicação para uso em algoritmos de recuperação desenvolvidos paralelamente no Grupo de Tolerância a Falhas da UFRGS, não estava disponível a implementação de aplicações que atingissem este objetivo específico.

Um dos pontos impostos inicialmente foi a busca de aplicações disponíveis - em contraste ao desenvolvimento próprio de aplicações. A escolha desta segunda opção - fazer uso de *software* desenvolvido por terceiros, é eliminar assim eventuais fatores que poderiam fazer com que os experimentos gerassem informações tendenciosas a algum dos enfoques, prejudicando a análise e a obtenção de resultados neutros. Então, partiu-se para a busca de programas desenvolvidos por terceiros.

Considerando o exposto, e como a análise a ser realizada está baseada na captura de mensagens, utilizaram-se duas aplicações distribuídas denominadas respectivamente de Ping-Pong e Escolha de Líderes. Ambas têm como característica geral a troca de mensagens usando a comunicação por meio de *sockets* com TCP.

Outra característica que contribuiu para o uso dessas duas aplicações é o fato de que elas possuem algumas diferenças importantes tornando a análise mais ampla. Sabe-se que o comportamento e o tipo de ações realizadas pelas aplicações pode ter um

impacto significativo nos resultados. A aplicação Ping-Pong possui uma troca de mensagens bastante intensa, o que ocasiona um número elevado de chamadas de sistema (*socketcall*), e na aplicação Escolha de Líderes este número é menor.

4.3.1 Algoritmo de Escolha de Líderes

Este algoritmo consiste de um problema de escolha de um elemento destacado, em um sistema onde vários processos executam tarefas semelhantes. Em determinado momento, pode ser necessário que seja eleito um novo líder para um grupo de processos quaisquer. Para realizar a escolha, o algoritmo utilizado é o proposto por Lynch [Lyn 96]. A idéia básica do algoritmo é exposta a seguir:

> Dado um conjunto de processos com uma topologia de interconexão qualquer, eles necessitam trocar N mensagens com os seus vizinhos para identificar o novo líder; onde N representa o diâmetro do grafo que representa a topologia de interconexão dos processos. O novo líder será eleito de acordo com o identificador do processo em questão, sendo que o processo com o maior ID será eleito.

A seguir encontra-se uma descrição simplificada do algoritmo:

```
Processo i:
max = seu_id;
Para i= 1 até n faca
   Para todos os vizinhos faca
      Send(max)
   fim_para;
   Receive(aux);
   max = máximo(aux, max);
fim_para;
Se id=max então <sou líder> senão líder = max;
```

O problema de Escolha de Líderes foi implementado utilizando-se a linguagem Java. Os canais de comunicação entre processos foram feitos com *sockets* usando TCP. Os dois processos principais do problema são: o gerador e os nodos. O gerador encarrega-se de criar randomicamente uma matriz de interconexão qualquer. Esta matriz é simétrica, não reflexiva e representa um grafo não desconexo. O gerador espera todos os nodos conectarem-se a ele e logo após envia para cada um dos nodos a informação sobre os seus vizinhos. Logo após, ele morre e os nodos começam a executar o algoritmo propriamente dito. No nodo, existe uma *thread* auxiliar que representa o servidor. Nesta *thread*, ele fica esperando por N conexões de *sockets* de outros processos vizinhos, onde N é o número de vizinhos já informados pelo gerador. As conexões geradas, apesar de serem bidirecionais, são utilizadas apenas para receber mensagens (*receive*). Por outro lado, na unidade de execução principal do processo, ele primeiramente tenta se conectar com todos os seus vizinhos, através da *thread* servidora de cada um deles, e essas conexões são utilizadas apenas para enviar mensagens (*send*).

4.3.2 Algoritmo Ping-Pong

O algoritmo Ping-Pong, conforme o seu próprio nome sugere, é um algoritmo bastante simples. Este parte da idéia que se possuam dois objetos, sendo um servidor e outro cliente. O servidor deve aguardar pela comunicação do cliente; nesta comunicação, ocorre a troca das palavras "ping" ou "pong", ou seja, quanto o servidor recebe a palavra "ping", ele envia para o cliente a palavra "pong". Isso também ocorre com o cliente quando recebe a resposta do servidor, ambos ficam substituindo uma palavra pela outra até uma determinada condição que interrompa esta comunicação.

Foi adotada a linguagem Java para implementar este algoritmo. O canal de comunicação utilizado para conexão entre os processos é feito através de *sockets*. O processo-servidor trata de criar um canal e aguarda pela conexão do processo-cliente.

5 Desenvolvimento dos protótipos e avaliação das abordagens

Neste capítulo, serão abordados os experimentos realizados e os resultados alcançados através destes. Os experimentos tiveram como ponto inicial as abordagens classificadas por Felber [FEL 98a], que são: integração, serviço e interceptação. O presente capítulo tem por objetivo apresentar e justificar as decisões tomadas durante este trabalho, que englobou um estudo das abordagens citadas acima e a implementação de protótipos para viabilizar uma análise quanto aos aspectos quantitativos e qualitativos.

A análise qualitativa tem com parâmetros os aspectos propostos e analisados por Felber, tais como modularidade, transparência e facilidade de uso. Os aspectos quantitativos dizem respeito à verificação de parâmetros sobre as abordagens que possam ser representadas numericamente como, a interferência sobre o desempenho normal da aplicação, provocado pela agregação de uma determinada abordagem.

Enfim, este capítulo tem por objetivos apresentar a forma e os meios utilizados na obtenção dos dados e os resultados obtidos a partir destes, gerando um conjunto de informações que possibilitem uma caracterização das diferentes abordagens, quanto a seus aspectos quantitativos e qualitativos.

5.1 Aspectos da análise qualitativa e planejamento da análise quantitativa

Para que a **análise qualitativa** realizada sobre as abordagens a partir da implementação de seus respectivos protótipos seja mais bem compreendida, deve-se previamente ter o conhecimento dos diferentes aspectos que compõem esta análise, que são [FEL 98a]:

- **Transparência** - é avaliada considerando a capacidade de adicionar novas funcionalidades à aplicação, sem que isso interfira diretamente na implementação desta aplicação;

- **Facilidade de uso** - este é um aspecto relevante que proporciona agilidade no desenvolvimento e pode possibilitar mais robustez e confiabilidade a aplicação, minimizando a possibilidade de erros durante o processo de implementação;

- **Portabilidade** - é o grau de independência em relação a uma determinada arquitetura apresentada por um determinado componente de *software*;

- **Modularidade** - aspecto que representa a capacidade de redução da complexidade de um sistema, dividindo-o em um conjunto de componentes acoplados livremente. Esses componentes podem ser agrupados de forma independente, o que possibilita uma redução de complexidade. A modularidade é um aspecto difícil de avaliar, pois envolve considerações de

implementação. Uma forma mais prática de avaliar este aspecto é seguir uma regra geral: componentes separados são mais modulares que os monolíticos.

Assim como os aspectos citados anteriormente, antes de descrever a **análise quantitativa** dos resultados é importante ter-se o conhecimento dos fatores que foram considerados nesta análise, ou seja, o planejamento adotado na coleta dos dados:

- o planejamento estratégico é composto dos fatores que oferecem suporte à geração e a captação dos dados que serão objeto da análise;

- o planejamento tático é descrição da forma como foi realizada a coleta dos dados, assim como a elaboração dos resultados.

Faz parte do **planejamento estratégico** a utilização de duas aplicações distribuídas: uma delas denominada de Ping-Pong e a outra de Escolha de Líderes. Ambas têm como característica geral a troca de mensagens utilizando a comunicação através de *sockets* com TCP. Essas aplicações foram descritas com um maior nível de detalhes na seção 4.3. Outros fatores considerados neste planejamento são o suporte de *hardware*, rede e, sistema operacional, já detalhados na seção 4.2 e apresentados de forma resumida na tabela 5.1.

TABELA 5.1 - Configuração do ambiente de suporte

Hardware	Rede	Sistema Operacional
▪ Pentium MMX; ▪ CPU 200MHz; ▪ Memória RAM 64Mb.	▪ Ethernet de 10Mbit/s	▪ Linux *kernel* versão 2.2.

O **planejamento tático** definiu os experimentos realizados com as diferentes abordagens propostos para execução de forma segmentada, ou seja, a coleta dos dados foi dividida em várias sessões[10]. Esta segmentação tem como objetivo minimizar a interferência de fatores externos sobre os resultados. Os experimentos obedeceram ao seguinte roteiro:

- os experimentos foram divididos em seqüências de vinte execuções; ou seja, primeiramente foi executada por 20 vezes a aplicação pura (sem mecanismos de captura);

- logo após, foram realizadas as demais seqüências, usando o protótipo da respectiva abordagem agregado à aplicação.

Essas sessões foram realizadas em dias distintos, até reunir um total de 60 (sessenta) amostras para cada abordagem. Dessas 60 (sessenta) amostras, foram utilizadas apenas 50 (cinqüenta). A decisão por usar 50 amostras deve-se à necessidade de realizar cálculos de desvio-padrão e intervalos de confiança[11] das amostras. Esses cálculos servem de indicadores de que as médias obtidas das amostras representavam,

[10] Uma sessão corresponde a uma seqüência de vinte execuções da aplicação.
[11] O intervalo de confiança tem como objetivo indicar um percentual de quão próxima à média das amostras esta da média, sendo que a utilização deste parâmetro só se justifica em um número de amostras acima de trinta [JAI 91].

de modo satisfatório, o fenômeno (o grau de interferência dos diferentes protótipos quando acoplados à aplicação).

A primeira aplicação a ser utilizada foi o **Ping-Pong**. A escolha baseou-se na simplicidade do algoritmo e do código-fonte, o que representa uma redução da complexidade do problema sem qualquer prejuízo para os resultados. Essa redução possibilitaria uma facilidade maior para a implementação e experimento dos protótipos, principalmente quanto à implementação da abordagem de serviço, que até então não se sabia se seria ou quanto seria necessário alterar da aplicação. Outro fator relevante para o uso do Ping-Pong é a sua característica de possuir uma troca intensa de mensagens, o que ocasiona um número elevado de chamadas de sistema (*socketcall*).

Esta aplicação é composta por dois processos chamados respectivamente de "pingpongServer" e "pingpongClient". Seguindo esta característica, os experimentos foram realizados em dois computadores. As sessões de tomada de tempo foram realizadas somente no computador com o processo pingpongClient. Nos experimentos realizados com o Ping-Pong, foram usadas sete configurações diferentes na aplicação. Na primeira, a aplicação era executada ciclicamente, repetindo-a por 400 (quatrocentas) vezes, onde o cliente apresentava um total de 2406 chamadas de sistema *socketcall*. Na última configuração, o ciclo foi ajustado em 1000 (um mil) vezes, alcançando um total de 6006 chamadas. O objetivo dessas diferentes configurações é verificar alterações no comportamento das diferentes abordagens.

Nos experimentos com a aplicação **Escolha de Líderes**, usou-se uma seqüência semelhante à aplicada no Ping-Pong, respeitando-se as peculiaridades desta nova aplicação. A Escolha de Líderes é composta por dois processos principais, que são gerador e nodos. No primeiro experimento, utilizou-se um processo-gerador que foi executado em um dos computadores e três nodos que foram executados em outros três computadores com a mesma configuração do usado pelo gerador. A cada execução, a tomada de tempo foi realizada em apenas um dos nodos. Na segunda configuração da Escolha de Líderes, o gerador continuou sendo executado no mesmo computador e os quatro nodos foram executados em outros computadores, com a mesma configuração do que estava executando o gerador. O nodo que estava sob avaliação permaneceu no mesmo computador que já havia sido usado na configuração do experimento anterior. No último experimento com essa aplicação, foram utilizados cinco nodos, e a seqüência do experimento seguiu o mesmo procedimento.

Vale ressaltar que, até se chegar ao planejamento descrito acima, e conseqüentemente aos resultados, numerosos experimentos foram realizados e seus resultados avaliados. Estas avaliações foram realizadas objetivando a obtenção de resultados confiáveis e satisfatórios. Quando se realiza experimentos em ambiente real, a complexidade torna-se elevada. O número de problemas a ser enfrentado é grande, devido a essa exposição ao mundo real, onde ocorrem problemas de *hardware*, como uma placa de rede defeituosa que pode ocasionar a perda de mensagens, por exemplo, e problemas de *software*, como o efeito do escalonamento sobre os experimentos. Esses fatos, se não avaliados com cuidado, podem comprometer os resultados.

5.2 Abordagem de interceptação, protótipo e análise.

A obtenção de informações por meio da abordagem de interceptação tem como sua principal característica a captura das informações em nível do usuário, isto é, como uma camada entre o sistema operacional e a aplicação. Com o conhecimento conceitual e o estudo sobre o projeto Eternal [NAR 97a, NAR 97b, NAR 99], apresentado por Felber em sua tese [FEL 98a], como exemplo de projeto que usa a abordagem de interceptação, direcionou-se os estudos no sentido de identificar e analisar os mecanismos disponíveis para a implementação do protótipo.

Neste primeiro momento, buscava-se referência sobre os aspectos da implementação da abordagem de interceptação, tendo-se como ponto de partida a classificação de Felber [FEL 98a]. Atualmente, este é um ponto que pode ser identificado como a primeira dificuldade superada. Isso se tornou relevante devido à expressiva demanda de tempo até a sua solução, e ao que ela proporcionou à pesquisa. A dificuldade refere-se à **variedade de nomenclatura** usada para denominar os mecanismos utilizados na interceptação. Com base na classificação de Felber, inicialmente buscava-se identificar referências através da palavra chave interceptação (*interceptation*) ou (*interceptation approach*), o que resultou na dificuldade de encontrar referências, esta foi superada quando se optou por procurar por chamadas de sistemas (*system calls*), encontrando-se várias referências que apresentam termos como: interposição (*interposition*), extensão (*extention*), aumentar (*enhance*), que se fazem valer do conceito de interceptação, como é o caso dos projetos Ufo [ALE 98a], Janus [WAG 99], Consh [ALE 98b]

Superada a dificuldade de encontrar informações sobre os aspectos de implementação, objetivou-se identificar o ambiente que ofereceria suporte para o desenvolvimento do protótipo. Através do estudo de casos, foi possível verificar o suporte oferecido pelos sistemas operacionais baseados em Unix para a implementação da abordagem de interceptação. As possibilidades identificadas de capturar chamadas de sistema utilizadas para implementar esta abordagem, que se encontram disponíveis em nível do usuário e não exigem modificações no sistema operacional são [PET 95]:

- modificação do sistema de bibliotecas – esta alteração é feita pela troca do procedimento de chamadas de sistema por uma versão que examine os argumentos das chamadas de sistema, retorne valores e armazene o estado externo em variáveis. Mas isso exige que as aplicações sejam ligadas com essas bibliotecas especiais, o que pode ser uma desvantagem do método. Segundo algumas experiências este método oferece um nível razoável de desempenho e transparência.

- interfaces de monitoramento - este método apresenta vantagens como trabalhar totalmente transparente não sendo necessária nenhuma modificação no binário da aplicação. Uma desvantagem é a perda de desempenho devido à troca de contexto para todas as chamadas de sistema. Adicionalmente, o sistema de arquivos `/proc` e a chamada de sistema de monitoramento *ptrace* possuem muitas variantes entre os sistemas Unix, dificultando a portabilidade.

5.2.1 Desenvolvimento do protótipo

Em seguida, veio a definição de usar o sistema operacional Linux como ambiente de suporte para implementação do protótipo, tendo como base desta opção os requisitos descritos no capítulo 4 e o estudo de casos. Através do estudo de casos, verificou-se a variedade de projetos que utilizam os "ganchos" (*hooks*) oferecidos pelo sistema operacional para desenvolver a interceptação das chamadas de sistema e com isto adicionar numerosas funcionalidades, como é o caso do Eternal [NAR 97a, NAR 97b, NAR 99] e vários outros referenciados nas sessões 2.5 e no capítulo 3.

Objetivando atender de modo satisfatório alguns critérios como: simplicidade, facilidade de uso, portabilidade, e um menor grau possível de alteração junto a aplicação chegou-se a dois mecanismos: o sistema de arquivos virtuais `/proc` e a chamada de sistema de monitoramento *ptrace*. A primeira necessidade de tomada de decisão referente à implementação surgiu no momento de definir qual destas interfaces de monitoramento melhor adequar-se-ia ao desenvolvimento do protótipo. Na literatura, encontraram-se afirmações como: o sistema de arquivos `/proc` é a forma mais elegante e eficiente de interceptar chamadas em sistemas Unix [VAH 96]; em sistemas como Solaris e Linux, o sistema de arquivos `/proc` provê acesso interno a cada processo em execução na máquina. Assim, cada processo pode ser tratado via interface-padrão, permitindo que as chamadas de sistemas sejam capturadas [NAR 99]; assim como as possibilidades apontadas por Petri & Langendöfer [PET 95] e descritas no início da seção 5.2.

Estas afirmações, juntamente com o estudo de projetos que tinham a sua implementação baseada no Solaris utilizando-se do `/proc` como Eternal [NAR 99], Ufo [ALE 98a], levaram a direcionar a pesquisa no sentido de escolher o sistema `/proc` para esta abordagem. Em uma análise mais detalhada sobre este mecanismo, incluindo a análise do código-fonte do Ufo, objetivando obter um melhor conhecimento para usá-lo na implementação do protótipo, possibilitou identificar a existência de diferentes semânticas do sistema `/proc`, quando se comparam os sistemas baseados em Unix. Esta diferença de semântica resulta nas múltiplas implementações do Ufo e do Janus, objetivando a sua portabilidade entre os sistemas operacionais Solaris e Linux. O Ufo, por exemplo, em sua versão para Solaris usa o sistema de arquivo virtual `/proc` e, em sua versão para Linux, usa a chamada de sistema *ptrace*, ambas com a finalidade de capturar chamadas.

Sistemas operacionais como o Solaris e OSF/1 oferecem uma interface específica, através do sistema de arquivos `/proc`, que provê uma forma de monitoramento e modificações de processos individuais [GOL 96, WAG 99, ALE 99]. A SVR4, versão compatível do Unix, é que dá suporte a este mecanismo, o mais poderoso para manipulação de processos em nível de usuário [JAI 2000, ALE 99]. O sistema operacional mantém um sistema de arquivos especial no diretório `/proc`. Cada entrada no diretório `/proc` é um arquivo, cujo nome é um número decimal correspondente ao identificador do processo (pid). Um processo-interceptador pode capturar e controlar outro processo operando no arquivo `/proc/pid` [ALE 99].

O sistema de arquivo virtual `/proc` provê o acesso a outro espaço de endereçamento de processo por meio de operações-padrões read() e write() e uma diversidade de requisições ioctl() [BER 98, RUB 99]. Ele é uma esperança de melhor desempenho, de controle mais sofisticado e uma boa chance de ser portável; então uma implementação completa de um sistema de arquivos `/proc` é raramente encontrada nos sistemas Unix em uso hoje [PET 98a]. No Linux, o sistema de arquivos virtuais `/proc`, possui a mesma estrutura de arquivos `/proc/pid`, mas com algumas diferenças de funcionalidade. É composto por um número de arquivos que é capaz de fornecer informações gerais do sistema, como uso de memória, estatística dos módulos carregados [BER 98], informações sobre o processo, mas não permite o controle sobre esses processos.

Esta análise mais detalhada sobre possíveis mecanismos de monitoramento do sistema Linux mostrou que a única primitiva de controle de processo em nível de usuário suportado por este sistema operacional é a chamada de sistema *ptrace* [WAG 99]. Esta chamada possibilita que um processo possa controlar outro e que o processo sob controle possa ter seus dados lidos e modificados. Isso é possível respeitando a política de segurança do sistema, ou seja, ele só viabiliza o controle de um outro processo caso ambos pertençam ao mesmo grupo e usuários [BER 98]. A chamada *ptrace* é a única interface de monitoramento encontrada em todos os sistemas do tipo Unix [PET 98a]; mas vale ressaltar que existem variações entre os diferentes sistemas.

Embora, ela seja usada na implementação de vários programas de depuração oferecidos com a maioria dos sistemas operacionais Unix, dentre estes se encontram o `strace, ltrace, trace, gdb` e outros [WAG 99]. A chamada *ptrace* apresenta algumas características que podem ser vantagens ou desvantagens, dependendo da finalidade ao qual a implementação se destina, como é o caso da granulosidade grossa [BER 98, WAG 99], que significa que ele captura todas as chamadas de sistemas ou nenhuma. Isso contribui para reduzir o desempenho da aplicação na qual este mecanismo está sendo aplicado; em contra partida, possibilita maior controle sobre a totalidade das chamadas, sem a necessidade de um grande esforço de programação. Esta característica não é apresentada pelo sistema de arquivos `/proc`, pois a captura de chamadas por esta interface é mais flexível, através de uma granulosidade fina. Outra característica é a capacidade de se ter acesso aos dados em seu pré-processamento (entrada) e em seu pós-processamento (saída). Isso pode ser um problema caso não seja necessário explorar essa capacidade, pois a chamada sempre realiza essa parada [PET 98a]. Aspectos referentes ao perfil e uso da chamada de sistema *ptrace* foram abordados mais detalhadamente na seção 4.1.5.

Com a identificação e a definição do mecanismo a ser usado, a especificação do protótipo teve como requisito o desenvolvimento de um protótipo tendo como única funcionalidade a captura de uma determinada chamada de sistema de um processo específico. Os resultados desta implementação objetivam gerar informações necessárias para formar o perfil da abordagem. Para tanto, o protótipo realiza a captura da chamada de sistema de comunicação interprocessos: a chamada *socketcall*.

Na implementação do protótipo optou-se pelo uso da requisição `PTRACE_ATTACH`, descrita na seção 4.1.5 de modo mais detalhado, na tentativa de simplificar a implementação do mesmo. A chamada de sistema *ptrace* usando esta

requisição tem a capacidade de tornar qualquer processo seu processo-filho. Este por sua vez, ao ter seu *flag* de depuração ativado e receber um sinal `SIGSTOP`, que interrompe sua execução, torna-se um processo sob controle de um novo pai, aqui denominado interceptador. Após o processo-interceptador executar a requisição `PTRACE_ATTACH`, anexando o processo desejado, toma-o sob seu controle. O processo-interceptador entra em uma execução cíclica (*loop*) onde a chamada de sistema *wait* é usada pelo interceptador para aguardar por um sinal enviado pelo processo-filho e verificar se este está parado.

No passo seguinte, é usada a macro da chamada *wait* `WIFSTOPPED` para verificar se o processo está parado, juntamente como a macro `WSTOPSIG`, também da chamada *wait*, para verificar qual o sinal que causou a parada do processo-filho. Com o processo-filho parado, usa-se novamente a chamada de sistema *ptrace*, agora com a requisição `PTRACE_GETREGS`, que possibilita a leitura dos registradores do processo-filho. Estes registradores são encontrados na estrutura contida no arquivo user.h, a estrutura pt_regs ou user_regs_struct, dependendo da versão do Linux, possibilita, entre outras informações, identificar qual a chamada que está sendo realizada pelo processo. Em seguida, o número da chamada capturada é atribuído a uma variável, e então são realizados os testes para identificar se é a entrada de uma chamada para o *kernel* ou o retorno desta.

Caso o processo não esteja parado pelo sinal `SIGTRAP`, utilizam-se novamente as macros da chamada *wait* para identificar o sinal com o objetivo de tratá-lo através do programa, com a finalidade de não bloquear a execução do processo, repassando este sinal para o processo-filho.

Após os procedimentos de testes citados acima, é realizado o tratamento de seus resultados. O primeiro caso (LT_EV_SIGNAL) a ser tratado é quando o sinal recebido não é um `SIGSTOP`. Este sinal que foi anteriormente armazenado é repassado para o processo-filho por meio da requisição do *ptrace* a `PTRACE_SYSCALL` para continuar o processo. Na segunda opção (LT_EV_EXIT), é feita apenas a identificação de que o processo-filho recebeu um sinal de termino de sua execução. Nas duas opções seguintes (LT_EV_SYSCALL/_SYSRET), é onde se pode trabalhar com as informações da chamada capturada. Nessas opções, se tem a identificação da chamada que está sendo realizada, assim como a possibilidade de trabalhar a chamada antes de sua entrada no *kernel* ou em sua saída. No protótipo, conforme seu objetivo, foi realizada somente a identificação da chamada *sockcall* na opção que antecede a entrada da chamada no *kernel*. Para finalizar as opções que tratam do processo, há uma opção *default* onde é exibida uma mensagem para eventos desconhecidos.

5.2.2 Análises qualitativa e quantitativa dos experimentos de interceptação

A **análise de ordem quantitativa** tem como objetivo atender a necessidade de dispor de informações de ordem quantitativa, ou seja, numéricas, sobre a abordagem de interceptação. Conforme exposto junto à seção 5.1, a obtenção destas informações provém da realização de experimentos, utilizando o protótipo da abordagem agregada as diferentes aplicações. As aplicações empregadas para a realização destes experimentos

apresentam diferentes características: a aplicação Ping-Pong representa troca intensa de mensagens, e a aplicação Escolha de Líderes representa troca de mensagens em menor escala. Essas aplicações foram descritas de forma mais detalhadas na seção 4.3.

Os resultados constantes nas tabelas 5.2 e 5.3 reúnem dados quantitativos dos experimentos descritos no planejamento estratégico e tático na seção 5.1. Estas tabelas apresentam os resultados dos experimentos feitos com sete configurações diferentes da aplicação Ping-Pong e com três diferentes configurações da aplicação Escolha de Líderes.

Para que as medidas de tempo realmente demonstrassem de modo confiável os experimentos, utilizaram-se alguns recursos estatísticos, como é o caso do cálculo do desvio-padrão das amostras e do cálculo do intervalo de confiança. O intervalo do desvio-padrão dos experimentos com a aplicação Ping-Pong estende-se entre os percentuais de 4,85% a 9,75%. Nos experimentos com a aplicação Escolha de Líderes, o intervalo está entre os percentuais de desvio-padrão de 9,64% a 13,80%.

O desvio-padrão, juntamente com intervalo de confiança de cada uma das configurações dos experimentos, permitiu a identificação de um intervalo no percentual de interferência causado pela abordagem de interceptação. Na aplicação Ping-Pong o intervalo fica entre 0,93% a 6,89% e na Escolha de Líderes, entre 2,09% a 4,74%. Diante da possibilidade de se agregar novas funcionalidades e dos benefícios que estas podem trazer à aplicação pode-se dizer que os percentuais de interferência são suportáveis pela aplicação.

TABELA 5.2 - Análise da abordagem de interceptação na aplicação Ping-Pong

Experimentos / nº Chamadas	**TM* sem mecanismo (ms)**	**TM* com mecanismo (ms)**	**Interferência (%)**	**D. Padrão (%)**	**Intervalo de Confiança[12] de 95% para μ**	
PP400 / 2406	14973	15981	6,73	6,03	15629,9	16331,4
PP500 / 3006	18277	19535	6,89	8,82	18779,9	20290,3
PP600 / 3606	22170	23332	5,24	8,23	22644,7	24019,8
PP700 / 4206	26034	27323	2,72	9,75	26156,0	28490,0
PP800 / 4806	29044	29313	0,93	7,57	28519,5	30106,7
PP900/ 5406	32269	35930	6,27	6,66	34880,5	36978,8
PP1000/6006	36299	37497	3,30	4,85	36835,6	38158,5

(**TM***) Tempo Médio de Execução

TABELA 5.3 - Análise da abordagem de interceptação na aplic. Escolha de Líderes

Experimentos /nº Chamada	**TM* sem mecanismo (ms)**	**TM* com mecanismo (ms)**	**Interferência (%)**	**D. Padrão (%)**	**Intervalo de Confiança[12] de 95% para μ**	
EL 3/ 52	3918	4068	3,83	13,80	3863,2	4241,6
EL 4/ 75	28035	28807	2,09	9,94	27552,1	30060,9
EL 5/ 80	30297	32098	4,74	9,64	30741,8	33453,9

(**TM***) Tempo Médio de Execução

[12] Os valores dispostos no Intervalo de Confiança são os valores extremos.

A segunda análise apresentada nesta seção é a **análise de ordem qualitativa**, realizada durante a implementação do protótipo de interceptação. Esta análise tem como propósito identificar e verificar os resultados descritos por Felber [FEL 98a], segundo os aspectos qualitativos, citados no início deste capítulo.

O primeiro aspecto a ser avaliado no protótipo de interceptação, que faz uso da chamada de sistema *ptrace* para implementar a extensão, é a **transparência**. Este é um aspecto que foi constatado claramente durante a implementação do protótipo, devido à capacidade que a chamada de sistema de monitoramento *ptrace* tem de possibilitar a adição de novas funcionalidades á aplicação, sem que haja a necessidade de qualquer alteração no código da aplicação ou aumento da complexidade na implementação desta aplicação.

Além do protótipo que aborda especificamente o aspecto da capacidade de captura das chamadas, também pode verificar se a adição de diferentes funcionalidades em projetos como o Ufo [ALE 99] e Janus [WAG 99]. Eles fazem uso desta chamada para beneficiar-se de aspectos como a transparência. Um aspecto que deve ser considerado, embora não contido na análise de Felber, é a condição de manter-se "discreto", ou seja, quando agregada à aplicação, a chamada *ptrace* não é de fato totalmente transparente, pois ela apresenta interferência no desempenho da aplicação. Isto é importante quando se busca um mecanismo que possua a capacidade de adicionar novas funcionalidades com uma atribuição mínima de carga sobre a aplicação.

A **facilidade de uso** é outro aspecto que se pode verificar durante o desenvolvimento do protótipo. A própria transparência oferecida por este mecanismo já traz consigo a atribuição de facilidade de uso no desenvolvimento das aplicações. A utilização da chama *ptrace,* juntamente com uma interface bem-definida, pode adicionar características como reusabilidade, transparência, entre outros, proporcionando a implementação de funcionalidades que permitam mais robustez e confiabilidade à aplicação, minimizando a possibilidade de erros durante a implementação. Vale ressaltar que quando se considera a implementação propriamente dita, o uso da chamada de sistema *ptrace* deixa muito a desejar sob este aspecto, pois apresenta uma péssima documentação, gerando a necessidade de se buscar informações sobre o uso de suas requisições nos códigos fonte de ferramentas como *ltrace*, *strace*, tornando a utilização bastante difícil.

Outro aspecto verificado através da implementação do protótipo foi a **portabilidade**, que demonstra a capacidade do componente ser independente da arquitetura. O protótipo beneficiou-se em sua implementação de características de baixo nível do sistema operacional, como é o caso da chamada *ptrace.* Conseqüentemente, tornou-se dependente deste sistema, e, portanto, não oferece o que se pode chamar de uma portabilidade direta. Para que possa prover este aspecto na prática, tornando-se portável entre os demais SO baseados na plataforma *Unix*, é necessária a utilização de recursos da linguagem como as diretivas do pré-processador (#if, #ifdef, etc), que proporcionam uma compilação condicional [SCH 90].

O último aspecto analisado foi a **modularidade**. A modularidade é um aspecto difícil de avaliar, pois envolve considerações de implementação. Uma forma mais prática de avaliar este aspecto é seguir uma regra geral. Componentes separados são mais modulares que os monolíticos. Seguindo essa regra, o conceito da abordagem de interceptação e as características do mecanismo usado na implementação do protótipo, pode-se afirmar que a modularidade é um aspecto presente. Para tanto, o mecanismo viabiliza que as funcionalidades implementadas com o seu uso sejam separadas da aplicação. Deste modo, provê um componente livre, que pode ter suas funcionalidades alteradas e incrementadas sem a necessidade de alterar a aplicação.

5.3 Abordagem de Integração, Protótipo e Análise

A idéia chave da abordagem de integração é fazer com que as novas funcionalidades e o processamento sejam suportados pelo ambiente (*middleware*), através da modificação ou da construção de mecanismos de processamento para a finalidade visada. Isto vem ressaltar a importância da escolha do ambiente que dará suporte à implementação. O ambiente deve oferecer suporte às novas funcionalidades ou a possibilidade de se deixar modificar. A importância da escolha ou definição do ambiente foi um aspecto sempre presente, a partir do momento que se iniciou as pesquisas buscando identificar as características referentes à implementação desta abordagem e um ambiente que pudesse suportar a inclusão de novas funcionalidades junto a seu código.

Os sistemas operacionais baseados em Unix parecem ser os mais indicados, por permitirem o uso de módulos (*device drivers*) [RUB 99]. Adicionalmente, aliou-se o fato de que haveria uma certa uniformidade do ambiente de suporte entre as abordagens de integração e interceptação, considerando o propósito comparativo da implementação de ambos. Outro fato que veio reforçar a idéia de usar os recursos oferecidos pelo sistema operacional, foi a localização de projetos como SILC [GHO 98] e *kernel hypervisor* [MIT 2000], que fazem uso do suporte a módulo oferecido pelos sistemas operacionais baseados em Unix para implementar novas funcionalidades.

Na investigação do suporte a módulo oferecido pelo Linux, percebeu-se o potencial que esta característica do sistema oferece, como a possibilidade de usar módulos para capturar chamadas de sistemas em seu pré ou pós-processamento, como um módulo adicional de controle de segurança, ou prover suporte a replicação [MIT 2000]. Enfim, é um recurso poderoso disponibilizado pelo sistema, o qual oferece todos os requisitos necessários para o desenvolvimento do protótipo da abordagem de integração. Cabe ressaltar que, para utilizar este recurso disponibilizado pelo sistema, é exigido o mínimo ou nenhum tipo de alteração no sistema operacional existente [GHO 98].

5.3.1 Desenvolvimento do protótipo

Conforme o exposto, a opção pelo uso de *device drivers* como mecanismo para a implementação do protótipo desta abordagem, tem como suporte a característica disponibilizada pelo sistema operacional, além de outros critérios que o uso de um módulo na implementação do protótipo satisfaz: a simplicidade considerando-se que

para o desenvolvimento do protótipo não será necessário o uso de muitos recursos, ou seja, alterar o sistema e a; facilidade de uso, pois um módulo pode ser carregado sem a necessidade de reinicializar o sistema operacional.

Implementar um módulo é uma tarefa de dificuldade média, desde que alguns pontos sejam observados. As informações neste parágrafo objetivam alertar para alguns destes pontos percebidos durante a fase de implementação, como também para alguns aspectos técnicos referentes à compilação de um módulo. O primeiro ponto, diz respeito aos problemas de compilação: como um módulo *kernel* não é um programa executável independente, mas um arquivo-objeto que será ligado a um *kernel* em execução, ele deve ser compilado com o parâmetro -c do compilador gcc[13]. Adicionalmente, todos os módulos *kernel* devem ser compilados com determinados parâmetros de pré-processador. Um exemplo disto é a necessidade de especificar -*MODULE* que identifica os arquivos de cabeçalho para oferecer as definições apropriadas para o módulo *kernel* [POM 99]. Outros detalhes que devem ser observados quando se opta por trabalhar em nível do *kernel,* é que não se tem acesso as facilidades oferecidas pelas bibliotecas ao nível do usuário. Outro cuidado é referente à questão das falhas, pois quando elas ocorrem neste nível, podem ser fatais, ao menos ao próprio processo. Estes e outros fatos tornam a programação neste nível um trabalho mais complexo para o programador que não tem esta tarefa como cotidiana.

Visando atender aos objetivos deste trabalho, a implementação do módulo tem como meta realizar a captura da chamada de sistema *socketcall,* sem adicionar qualquer outra funcionalidade ao sistema operacional ou à aplicação. Para isto, ele trabalha com a tabela de endereçamento das chamadas, a *sys_call_table*, alterando o endereço da função do *kernel* desejada para um outro onde se encontra a nova função. Esta troca de endereços é realizada no momento em que o módulo é carregado, junto à função *init_module(),* que tem a responsabilidade de chamar a *module_register_chrdev* para adicionar o dispositivo de *driver* (módulo) à tabela de dispositivos de caracteres [POM 99]. O processo inverso da troca de endereços é feito junto à função encarregada de informar ao *kernel* que o módulo não está mais disponível, a função *cleanup_module()* [POM 99, RUB 99].

Após a descrição de algumas informações importantes para o entendimento do funcionamento do protótipo, neste parágrafo será realizada uma descrição do desenvolvimento deste protótipo. O módulo implementado recebe a identificação do processo (*pid*), via linha de comando, através da função `MODULE_PARM (var,"tipo")`. Logo após, é criado um apontador para a função do *kernel*, que corresponde à chamada de sistema original (sys_*socketcall*). Como passo seguinte, criou-se uma nova função denominada de *our_sys_socketcall:* a esta função foi adicionada a verificação de qual processo está realizando a chamada de sistema. Caso seja o processo informado inicialmente, o contador colocado na função *our_sys_socketcall* é incremento e retorna chamando a função do *kernel* original, sem qualquer alteração em seus argumentos. Cabe ressaltar, que a função *sys_socketcall(int call,unsined long *args)*, que representa a chamada de sistema *socketcall* implementa várias funcionalidade através do seu primeiro parâmetro. Como por exemplo, a finalidade de prover a programação de *sockets*. Devido a isso, o contador implementado

[13] Todas as distribuições do Linux incluem o GCC, o compilador GNU para as linguagens C e C++. O comando para compilar arquivos de código fontes em C e C++ é o gcc.

junto à função *our_sys_socketcall,* foi definido como uma matriz, possibilitando que ao fim do módulo fosse possível obter-se o total de chamadas *socketcall*, assim como o número de vezes que cada funcionalidade foi executada.

5.3.2 Análises qualitativa e quantitativa dos experimentos de integração

A **análise de ordem quantitativa** também é realizada com a abordagem de integração, como parte da busca de informações que permitam a obtenção de uma ordem entre as abordagens. Visando atingir ao objetivo de modo consistente, os experimentos com esta abordagem seguiram os mesmos procedimentos dos experimentos com a abordagem de interceptação. Também foram mantidas as aplicações utilizadas nos experimentos, isto é, a aplicação Ping-Pong e a aplicação Escolha de Líderes.

Os experimentos com as aplicações Ping-Pong e Escolha de Líderes têm seus resultados demonstrados nas tabelas 5.4 e 5.5. A tabela 5.4 mostra os dados resultantes dos experimentos com a aplicação Ping-Pong em suas sete configurações. Nessa tabela os percentuais de desvio-padrão das amostras nas várias configurações da aplicação Ping-Pong e o intervalo de confiança acrescentam confiabilidade aos percentuais de interferência calculados nos diferentes experimentos. Esses experimentos revelam um intervalo de interferência da abordagem de integração que varia de 0,88%, em seu menor valor e de 6,47%, em seu maior grau de interferência.

TABELA 5.4 - Análise da abordagem de integração na aplicação Ping-Pong

Experimentos / nº Chamadas	**TM* sem mecanismo (ms)**	**TM* com mecanismo (ms)**	**Interferência (%)**	**D. Padrão (%)**	**Intervalo de Confiança de 95% para μ**	
Pp400 / 2406	14973	15220	1,65	7,06	14829,0	15611,6
Pp500 / 3006	18277	19458	6,47	7,83	18790,2	20126,6
Pp600 / 3606	22170	23097	4,18	8,53	22391,9	23801,2
Pp700 / 4206	26034	26600	2,18	11,98	25204,0	27996,4
Pp800 / 4806	29044	29299	0,88	7,37	28525,6	30071,6
Pp900 / 5406	32269	33810	4,77	8,39	32566,0	35053,1
Pp1000/6006	36299	36674	1,03	5,27	35970,1	37378,2

(**TM***) Tempo Médio de Execução

A tabela 5.5 mostra os resultados dos experimentos com a aplicação Escolha de Líderes, além dos valores calculados do desvio-padrão e do intervalo de confiança, que mostram a representatividade dos dados. Os percentuais de interferência resultantes dos vários experimentos com essa aplicação variam de 0,64% a 2,58%, sendo que os percentuais do desvio-padrão estão em um intervalo entre 7,55% a 14,50%.

TABELA 5.5 - Análise da abordagem de integração na aplicação Escolha de Líderes

Experimentos / nº Chamadas	**TM* sem mecanismo (ms)**	**TM* com mecanismo (ms)**	**Interferência (%)**	**D. Padrão (%)**	**Intervalo de Confiança de 95% para μ**	
El3 / 52	3918	4019	2,58	14,50	3806,8	4199,6
El4 / 75	28035	28216	0,64	7,55	27281,8	29149,3
El5 / 80	30297	30645	1,15	8,09	29558,2	31732,3

(**TM***) Tempo Médio de Execução

A **análise qualitativa,** nesta abordagem, também tem como base os dados coletados durante o processo de implementação do protótipo de integração.

A característica **transparência** pode ser atribuída a essa abordagem. O módulo implementado veio a comprovar isso, através da forma como ele provê a extensão do ambiente, viabilizando a adição de novas funcionalidades a diferentes aplicações, independente destas. Esta capacidade de oferecer suporte à implementação de diferentes funcionalidades pode ser constatada não só através do protótipo, mas também junto ao projeto *Kernel Hypervisor* [MIT 2000] que pode ser usado tanto para tornar um componente mais robusto, como também para agregar várias funções de segurança como auditoria. Estas funcionalidades podem ser implementadas sem a necessidade de alterar a implementação das aplicações. No que diz respeito à interferência, quando agregado à aplicação, o módulo mantém o desempenho da aplicação praticamente inalterado. Isto se refere à captura das chamadas, pois dependendo da funcionalidade a ser adicionada pode vir a alterar esta condição.

As propriedades que fazem parte do aspecto **facilidade de uso**, como prover agilidade no desenvolvimento, tornar a aplicação mais robusta e confiável, podem ser perfeitamente satisfeitos pelo uso de um módulo. Isso significa que a abordagem em si não cria empecilhos à manutenção da facilidade. A agilidade de desenvolvimento pode ser decorrente do desenvolvimento do módulo com característica de flexibilidade, com diversas funcionalidades implementadas. Por meio da implementação do protótipo e do estudo dos casos base para esta abordagem, pode-se identificar a presença deste aspecto. Um fato que é considerado como negativo do ponto de vista de um usuário comum (desconsiderando a questão da segurança do próprio sistema operacional), é que módulos só podem ser carregados e descarregados por um usuário que tenha permissão de *root*. Mas é importante lembrar que a carga e a descarga podem ser realizadas sem a necessidade de reiniciar o sistema operacional, o que vem a facilitar o seu uso. Quanto à facilidade de uso em relação a sua programação, ela é bem menos complicada que a utilização, por exemplo, da chamada *ptrace,* pois se encontra um material mais bem organizado em termos de informações de como trabalhar com módulos. Exemplos disto são o guia de Ori Pomerrantz [POM 99] e o livro de Alessandro Rubini [RUB 99]. Entretanto, deve-se considerar como dificuldade que não se pode contar com recursos como as bibliotecas disponíveis ao nível do usuário. Mas isso pode ser contornado através da implementação do capturador ao nível do *kernel* e das funcionalidades ao nível do usuário [GHO 98], caso o desempenho não seja um fator crucial para a aplicação, pois este tipo de solução acarreta em uma maior carga para a aplicação, devido à troca de contexto. Outra dificuldade esta relacionada ás diferentes versões do *kernel* que pode exigir alterações no código do módulo para que este funcione corretamente.

No aspecto **portabilidade**, que representa a independência da arquitetura, visto que o protótipo desenvolvido para implementar esta abordagem, utiliza-se de recursos de baixo nível oferecidos pelo sistema operacional tornou-se dependente deste, comprometendo a portabilidade direta. Logo, a portabilidade para outros sistemas operacionais não seria verificada totalmente, embora, este não fosse um item previsto inicialmente neste trabalho. Entretanto, há variações de implementação que podem definir resultados contraditórios na análise da questão de portabilidade. No estudo de

casos, ficou claro que existe a possibilidade de se desenvolver um módulo portável, no estudo do *Kernel Hypervisor* [MIT 2000], no qual a plataforma Linux também foi usada na implementação do projeto. O *Kernel Hypervisor* é portável para diferentes sistemas operacionais que possuam características afins, como o suporte à carga de módulo e acesso à estrutura de dados das chamadas de sistema. No caso analisado por Felber, a abordagem não apresenta portabilidade: o Electra [MAF 95a] altera o ORB para prover suporte a grupo, tornando-se dependente deste. O Electra pode ser portado para diferentes SO, mas ele não é portável no que diz respeito ao ORB.

A identificação da **modularidade** busca características como a possibilidade de reduzir a complexidade da aplicação, assim como possuir a capacidade de ser um componente livre; na prática, estas características são avaliadas pelo uso de componentes separados. Na integração, estas características que compõem o conceito da modularidade são disponibilizadas através do uso de um módulo. Ele provê a capacidade de implementar novas funcionalidades como módulos independentes da aplicação, possibilitando também que esta seja incrementada ou alterada sem influir na aplicação para tanto, mantendo a compatibilidade entre suas evoluções. Exemplos desta funcionalidade são os módulos de criptografia [GHOR 98], replicação [MIT 2000]. Enfim, funcionalidades podem ser implementadas objetivando oferecer propriedades tais como simplicidade, agilidade e robustez, no momento da implementação da aplicação.

5.4 Abordagem de Serviço, Protótipo e Análise.

A abordagem de serviço está fortemente ligada ao paradigma de orientação a objetos. Conceitualmente ela tem a sua estrutura composta por um conjunto de objetos de serviços, que têm como finalidade prover diferentes serviços [LUN 2000a]. Os objetos de serviço servem de bloco de construção (*building blocks*) para dar suporte ao desenvolvimento de várias aplicações como, por exemplo, oferecer tolerância a falhas por meio de replicação de objetos.

Esta abordagem apresenta uma idéia diferente das abordagens de integração e interceptação, na forma de disponibilizar para a aplicação o benefício de seus recursos. Para que isso ocorra, a aplicação deve ser projetada para tal, o que vem a significar mais definições e preocupações no momento de especificar a aplicação. Essa abordagem pode ser descrita como sendo um conjunto de objetos que não precisam necessariamente estar localizados em uma única máquina, o que dá robustez ao serviço, e que oferecem serviços variados. Para que estes serviços sejam transparentes utilizam-se objetos *proxies*.

5.4.1 Desenvolvimento do protótipo

O desenvolvimento do protótipo associado ao presente trabalho não seguiu exatamente o conceito da abordagem de serviço, no que se refere a ser composto por um conjunto de objetos. Este trabalho tem entre seus objetivos, verificar a interferência causada à aplicação, quando uma abordagem é utilizada para coletar informações sobre a implementação dos protótipos. Para tanto, não é necessário desenvolver um conjunto de objetos: basta desenvolver-se um objeto que seja responsável pela comunicação, pois

não está sendo avaliado o potencial quanto à extensão de novas funcionalidades, mas a captura da comunicação.

A abordagem de serviço pressupõe que as aplicações sejam desenvolvidas com este contexto em mente. No presente trabalho, entretanto, a adoção de aplicações prontas fez com que elas não possuíssem interfaces preparadas para utilização de um outro objeto, como seria correto para o uso da abordagem de serviço. Assim, o desenvolvimento do protótipo baseou-se nas características de implementação de cada uma das aplicações, diferentemente do modo como foi realizado o desenvolvimento dos outros protótipos, onde foi implementado um único, independente das aplicações usadas nos experimentos.

Ao iniciar os estudos sobre o código fonte da primeira aplicação, o Ping-Pong, percebeu-se que seria mais simples implementar um objeto que se adaptasse à aplicação, pois alterar o código da aplicação envolve o risco de causar alguma distorção, prejudicando os resultados. Neste caso específico, a implementação do protótipo foi bastante simples, após analisar o código da aplicação e verificar que ela é constituída de um objeto servidor e de um cliente. O protótipo foi implementado construindo-se outro objeto servidor com características de um *proxy*, sendo que a única funcionalidade dele era intermediar a comunicação entre os outros dois objetos. Após a implementação deste *proxy*, bastou indicar a porta do *proxy* para o servidor e a porta do cliente para o *proxy* e o protótipo de serviço estava implementado para a aplicação Ping-Pong.

A construção do protótipo para os experimentos com a aplicação Escolha de Líderes exigiu uma análise mais aprofundada da aplicação, devido ao algoritmo e a sua implementação. A aplicação Escolha de Líderes define a classe canal, a qual encapsula os objetos e dados necessários para estabelecer a comunicação de um nodo (classe Nodos) com seus vizinhos.

Para implementar a abordagem de serviço, especializou-se a classe canal em uma nova classe chamada canal1, que utiliza os métodos (enviaParaCliente e recebeDeCliente) definidos na classe canal. A classe canal1 é responsável pelo controle de comunicação da aplicação; com isto se viabiliza a captura das mensagens ao nível de aplicação, satisfazendo assim a concepção da abordagem de serviço.

5.4.2 Análise qualitativa e quantitativa dos resultados

A **análise de ordem quantitativa** da abordagem de serviço segue o mesmo roteiro das demais abordagens em seus experimentos e aplicações. Manter a uniformidade entre os experimentos é um fator importante quando se analisa um determinado aspecto, neste caso quantitativo, entre diferentes abordagens.

A tabela 5.6 ilustra os resultados dos experimentos com a abordagem de serviço com a aplicação Ping-Pong. Entre os percentuais de desvio-padrão obtido na análise quantitativa das três abordagens, com a aplicação Ping-Pong, a abordagem de serviço é a que possui o menor percentual, tendo se registrado valores no intervalo de 0,49% a 3,97%, podendo-se atribuir uma grande credibilidade aos resultados obtidos. Esta abordagem possui um grau de interferência bastante acentuado, tendo como grau

mínimo de interferência 80,32% e como grau máximo 98,60% percentuais que representam um custo bastante alto para a aplicação.

TABELA 5.6 - Análise da abordagem de serviço na aplicação Ping-Pong

Experimentos / nº Chamadas	TM* sem mecanismo (ms)	TM* com mecanismo (ms)	Interferência (%)	D. Padrão (%)	Intervalo de Confiança de 95% para μ	
Pp400/ 2406	14973	29117	94,47	3,97	28696,7	29588,0
Pp500/ 3006	18277	36209	86,09	2,74	35774,2	36344,6
Pp600/ 3606	22170	43497	96,20	2,92	43043,0	43951,2
Pp700/ 4206	26034	51030	86,77	1,56	50682,0	51378,3
Pp800/ 4806	29044	57683	98,60	1,59	57355,0	58011,5
Pp900/ 5406	32269	64787	80,32	1,42	64383,6	65190,5
Pp1000/6006	36299	72401	99,45	0,49	72271,1	72530,0

(**TM***) Tempo Médio de Execução

Nos experimentos com aplicação Escolha de Líderes, pode-se verificar um grau de interferência menor que o encontrado pela abordagem de serviço nos experimentos com a aplicação Ping-Pong. Na tabela 5.7, é mostrado que o grau de interferência causado pela a abordagem de serviço varia de 11,15% a 51,58% em seu grau de interferência máximo. O resultado obtido permite inferir que em aplicações que não possuem uma troca intensa de mensagens, a interferência causa em média um acréscimo de 50% no tempo de execução.

TABELA 5.7 - Análise da abordagem de serviço na aplicação Escolha de Líderes

Experimentos / nº Chamadas	TM* sem mecanismo (ms)	TM* com mecanismo (ms)	Interferência (%)	D. Padrão (%)	Intervalo de Confiança de 95% para μ	
El3 /52	3918	5938	51,58	6,60	5794,9	6060,6
El4 /75	28035	32019	11,15	12,88	30211,6	33826,4
El5 /80	30297	41297	28,66	9,98	39491,5	43102,8

(**TM***) Tempo Médio de Execução

Na realização da **análise qualitativa** sobre a abordagem de serviço, descrita abaixo, seguem-se os mesmos elementos de análise dos casos anteriores. Esta análise inicia-se pelo aspecto da **transparência**, que se refere à capacidade de dispor de uma funcionalidade cujas especificidades não precisaram ser consideradas no momento da implementação da aplicação. A abordagem de serviço dispõe de transparência; um exemplo desta é o OGS [FEL 98a], que provê o uso de grupo de objetos pela aplicação sem que a referência a este grupo seja uma preocupação para o desenvolvedor da aplicação.

Mas analisando especificamente o mecanismo (objeto), a adoção da abordagem de serviço não oferece transparência. No próprio conceito da abordagem, é possível perceber que, para fazer uso das funcionalidades oferecidas pelos objetos de serviço, há necessidade de que a aplicação seja desenvolvida referenciando explicitamente esses objetos. Esta relação entre a aplicação e os objetos de serviço ficaram evidente no desenvolvimento do protótipo, seção 5.4.1. Enfim, pode-se verificar que essa abordagem possui uma transparência quanto ao serviço disponibilizado pelas funcionalidades agregadas ao mecanismo, mas não quanto ao uso do mecanismo em si, isto é, na referência aos objetos de serviço.

A **facilidade de uso** é um aspecto que também está presente, ilustrando-se pela capacidade que a abordagem de serviço tem de prover agilidade no desenvolvimento através da reusabilidade. Os objetos de serviço servem de blocos de construção (*building blocks*) para dar suporte ao desenvolvimento de várias aplicações [LUN 2000a].

Por outro lado, a análise da facilidade de uso em relação implementação, ou seja, da construção de objetos de serviço, torna-se um pouco complexa, pois é possível realizá-la sob ângulos distintos. Sob o ângulo da aplicação, ou seja, quando uma aplicação que não foi projetada para usar os benefícios oferecidos pela abordagem de serviço, mas deve disponibilizar novas funcionalidade a essa aplicação por meio dessa abordagem. Este é um caso onde a facilidade de uso vai depender da forma como essa aplicação foi desenvolvida: se a programação foi realizada com interfaces bem-definidas, a construção dos objetos de serviço é realizada com uma maior facilidade. Caso contrário, pode dificultar o seu uso. Sob o ângulo dos objetos de serviço, ou seja, de construir um conjunto de objetos que ofereça diferentes funcionalidades, através de interfaces bem-definidas, para que aplicações a serem implementadas possam utilizar-se dessas funcionalidades, a facilidade de uso é um aspecto presente.

A **portabilidade** faz parte da abordagem de serviço, em seu projeto essa abordagem mostra-se composto por um conjunto de objetos dependentes dentre si, mas independentes do ambiente e da aplicação. Isso faz com que sua implementação tenha como base os recursos decorrentes do paradigma de orientação a objetos e fornecido pela linguagem de programação, tornando-a portável.

A **modularidade** é uma característica que faz parte da concepção dessa abordagem. Sendo composto por objetos de serviço que servem de bloco de construção para dar suporte ao desenvolvimento de várias aplicações, esta abordagem atende aos requisitos básicos que constituem a modularidade como forma de reduzir a complexidade no desenvolvimento da aplicação, e como a possibilidade de serem livremente acoplados e alterados.

5.5 Comparação das abordagens quanto à análise qualitativa

O primeiro aspecto avaliado da análise qualitativa foi a **transparência,** que teve suas características claramente constatadas, durante a implementação dos protótipos. Percebeu-se que existem diferenças quanto à forma como a transparência é enfocada: pode ser como uma capacidade de oferecer novas funcionalidades para a aplicação, ou na forma como a aplicação fará uso destas.

A implementação dos protótipos das abordagens de interceptação e integração utilizou a chamada de sistema *ptrace* e a criação de um módulo, como mecanismos de captura de chamadas. Verificou-se que, com a utilização desses mecanismos, a transparência pode ser avaliada sob duas formas: a funcionalidade oferecida (por exemplo, criptografia, replicação) e a forma como a aplicação fará uso do mecanismo de captura das chamadas. Ambas são facilmente alcançadas pela integração e interceptação. Para tanto, não há necessidade de qualquer conhecimento do uso do mecanismo por parte da aplicação. Conseqüentemente, as funcionalidades implementadas através desses mecanismos podem ser disponibilizadas para qualquer

tipo de aplicação, independentemente de ter-se ou não acesso ao código fonte da aplicação. A implementação do protótipo como um serviço, também permitiu verificar a presença da transparência no que se refere à funcionalidade. Um exemplo disso é o OGS [FEL 98]. Entretanto, quando esse mecanismo é utilizado para a aplicação beneficiar-se das funcionalidades oferecidas por ele, ela deve referenciar explicitamente o serviço. Portanto, nessa abordagem, a funcionalidade oferece transparência, mas o mecanismo de captura não.

A **facilidade de uso** é outro aspecto que se pode verificar durante o desenvolvimento dos diferentes protótipos. A própria transparência oferecida pelo modulo já apresenta a característica de facilidade de uso. O uso da chamada *ptrace* juntamente com uma interface bem-definida, pode adicionar outras características como reusabilidade, e transparência. Além disso, proporciona a implementação de funcionalidades que permitem, entre outros aspectos, robustez e confiabilidade da aplicação, minimizando a possibilidade de erros durante a implementação. Isso contribui para a facilidade de uso da abordagem de interceptação. Porém, deve-se ressaltar que quando se considera a implementação propriamente dita, a utilização da chamada de sistema *ptrace,* deixa muito a desejar. Esse mecanismo apresenta uma documentação deficiente, gerando a necessidade de se buscar informações sobre o seu uso nos códigos fonte de ferramentas como *ltrace*, e *strace*, o que dificulta a sua utilização.

O requisito de facilidade de uso pode ser perfeitamente satisfeito através do uso de um módulo. Esta técnica permite implementações flexíveis e garante uma boa agilidade de desenvolvimento. Dois exemplos disso são os projetos *Kernel Hypervisor* [MIT 2000]*,* e o SLIC [GHO 98]. Um fator negativo do uso de módulos é o de que só podem ser carregados e descarregados por usuários detentores de direitos de acesso em níveis de *root.* Em contra-partida, a carga e a descarga podem ser realizadas sem a necessidade de reinicializar o sistema operacional, o que vem a facilitar o seu uso. Em relação à sua programação, também é bem mais simples do que a utilização da chamada *ptrace*; e o material disponível é mais bem organizado. Uma dificuldade está na impossibilidade de usar recursos como as bibliotecas disponíveis em nível de usuário. Porém, isto pode ser contornado através da implementação do módulo de captura em nível de *kernel* e as funcionalidades em nível de usuário [GHO 98], caso o desempenho não seja um fator crucial para a aplicação.

Este é um aspecto que também está presente na abordagem de serviço, uma vez que os objetos de serviço servem como blocos de construção (*building blocks*) para suporte ao desenvolvimento das aplicações [LUN 2000a]. A análise de sua facilidade de uso em relação à implementação, realizada no desenvolvimento dos protótipos mostrou-se satisfatória. Foi possível identificar que as dificuldades potenciais dizem respeito a aspectos como a especificação e eventuais problemas com o uso da linguagem. Entretanto, ambas possuem uma boa documentação.

No aspecto **portabilidade,** novamente identificou-se uma similaridade entre os protótipos das abordagens de interceptação e integração. Ambos beneficiam-se dos recursos de baixo nível oferecidos pelo ambiente (sistema operacional), comprometendo assim a sua portabilidade. Isso não ocorre com a abordagem de serviço. A utilização da chamada *ptrace,* na implementação do protótipo da abordagem de interceptação, tem

como conseqüência a dependência do ambiente, e portanto não oferece o que se poderia chamar de uma portabilidade direta. Para melhorar este aspecto, é necessária a utilização de recursos da linguagem, como as diretivas do pré-processador, que proporcionam uma compilação condicional, tornando-se assim portável entre os diferentes sistemas operacionais baseados em Unix.

Assim como na interceptação, a integração implementada através do módulo também é dependente do ambiente, comprometendo o que foi chamado anteriormente de portabilidade direta. Embora não tenham sido efetuados testes práticos de portabilidade, pode-se verificar que há soluções aceitáveis para este aspecto, através do estudo do projeto *Kernel Hypervisor* [MIT 2000], que também utiliza a plataforma Linux na implementação de seu projeto. No relato ali contido, fica claro que, havendo características afins entre os sistemas, é perfeitamente possível desenvolver um módulo portável. Um exemplo dessas características é o suporte a carga de módulos que possuam acesso a estruturas de dados das chamadas de sistema. Diferente das anteriores, a abordagem de serviço mostra-se composta por um conjunto de objetos dependentes entre si, mas independentes do ambiente e da aplicação. Sendo um serviço independente do sistema operacional, essa abordagem tem sua implementação portável.

O último aspecto analisado, a **modularidade**, é de difícil análise porque está diretamente ligada a forma como é implementada a funcionalidade. Para esta análise, normalmente usa-se como regra geral a citada na seção 5.1: componentes separados são mais modulares do que os monolíticos. Seguindo esta regra, pode-se afirmar que a modularidade é um aspecto presente em todas as abordagens. Portanto, os diferentes mecanismos usados viabilizam que as funcionalidades implementadas sejam separadas da aplicação. Sendo assim, esses componentes livres podem ter suas funcionalidades alteradas e incrementadas, ou seja, podem oferecer diferentes funcionalidades objetivando prover mais simplicidade, agilidade e robustez, na implementação e no uso da aplicação.

5.6 Comparação das abordagens quanto à análise quantitativa

Através dos vários experimentos realizados, a análise quantitativa forneceu dados suficientemente confiáveis para que fosse possível a análise do grau de interferência que as abordagens causam à aplicação. Para tanto, foram usados recursos estatísticos, como o cálculo do desvio-padrão e do intervalo de confiança e também cuidados como os descritos junto ao planejamento estratégico e ao planejamento tático.

Os resultados dos experimentos que avaliaram a interferência das abordagens quando acopladas à aplicação, mostrados pelas tabelas 5.8 e 5.9, permitem classificá-las em uma ordem crescente quanto ao seu grau de interferência, que é a seguinte: integração, interceptação e serviço. Os dados não serviram somente para ordenar as abordagens. Analisando os percentuais dispostos pelas tabelas 5.8 e 5.9, é possível verificar que há apenas uma pequena diferença entre os percentuais de interferência das abordagens de integração e interceptação. Esta informação de uma certa forma veio a contrariar a idéia inicial, quanto apenas se possuía o conhecimento teórico das abordagens e dos mecanismos usados para a sua implementação: o módulo usado para implementar a integração e a chamada de sistema *ptrace* usada no protótipo da

interceptação. Pressupunha-se que o módulo, por ser carregado junto ao *kernel* do sistema, causaria maior interferência que a chamada *ptrace*.

TABELA 5.8 – Percentagem de interferência do Ping-Pong

Aplicação	Integração (%)	Interceptação (%)	Serviço (%)
Pp400	1,65	6,73	94,47
Pp500	6,47	6,89	86,09
Pp600	4,18	5,24	96,20
Pp700	2,18	2,72	86,77
Pp800	0,88	0,93	98,60
Pp900	4,77	6,27	80,32
Pp1000	1,03	3,30	99,45

TABELA 5.9 – Percentagem de interferência da Escolha de Líderes

Aplicação	Integração (%)	Interceptação (%)	Serviço (%)
EL3 52	1,89	4,03	49,68
EL4 75	1,25	2,75	14,59
EL5 80	0,57	0,64	32,36

A verificação da pequena diferença entre a integração e a interceptação altera a modo de avaliação dessas abordagens. A variação baseada apenas no conhecimento teórico, considerando a necessidade de optar por uma das abordagens para adicionar alguma funcionalidade a uma aplicação, onde o desempenho é um fator importante, certamente a escolha seria pela integração com o uso do módulo. O conhecimento que a implementação dos protótipos e os experimentos trouxeram, mudam esta avaliação prévia. Este conhecimento acrescenta na avaliação, além do fator de comparação grau de interferência, o tipo de funcionalidade a ser implementada. A diferença entre recursos disponíveis na integração, em nível de *kernel,* e a interceptação em nível de usuário como, por exemplo, as bibliotecas, fazem com que os aspectos de desenvolvimento e uso devem ser considerados. Assim, deverão ser considerados adicionalmente aspectos, diferenciados nos dois níveis referidos, como a diferença de complexidade e facilidade de manutenção e utilização.

As informações quantitativas serviram para identificar que, quando o desempenho é um fator crucial, a abordagem de serviço não é uma boa opção. Como é possível verificar nas tabelas 4.7 e 5.8, independente da configuração da aplicação usada no experimento, o percentual de interferência causado pela abordagem de serviço é bastante alto em relação às outras abordagens avaliadas. Mesmo quando considerada a sua menor interferência que é de 14,59% com o experimento realizado com a aplicação Escolha de Líderes, esse é um percentual bem considerável, pois é apenas para a captura da informação. Mas quando o desempenho não é um fator muito relevante, essa abordagem demonstra ser uma ótima opção, principalmente nos aspectos referentes à implementação e utilização. Para tanto, a abordagem de serviço possibilita o uso de todos os recursos que a linguagem utilizada na sua implementação disponibiliza, e trabalhando em um nível acima do sistema operacional a sua utilização é mais simples.

6 Conclusão

A idéia inicial deste trabalho surgiu da constatação da necessidade de utilizar informações das aplicações para realização de atividades implementadas em níveis inferiores. Conforme exposto na motivação do presente trabalho, alguns autores perceberam que essas informações da aplicação podem ser bastante interessantes na melhoria do desempenho dos sistemas. Além disso, a tolerância a falhas é incompleta sem o conhecimento e a realimentação das necessidades das aplicações.

Na busca de eficiência, os desenvolvedores de *software* se defrontam com desafios na busca de formas para obter essas informações, de modo que a interferência sobre a implementação das aplicações seja a mínima possível. O uso adequado das informações assim obtidas permite, por exemplo, o ajuste de parâmetros e a escolha de mecanismos adequados para a realização de atividades dos sistemas. Adicionalmente, aspectos tais como desempenho e segurança podem ser melhorados.

Neste contexto, publicações anteriores identificaram três enfoques básicos para obter-se informações referentes aos programas de aplicação: integração, interceptação e serviço. Neste trabalho, foram apresentados vários aspectos das análises qualitativa e quantitativa conseguidas pelo estudo do perfil dessas abordagens.

Nesta dissertação é analisado o comportamento desses mecanismos. Foram implementados protótipos, os quais possibilitaram a investigação de problemas relativos a estas técnicas. Os objetivos específicos reuniram a busca por informações qualitativas, tais como modularidade, transparência, facilidade de uso e portabilidade. Além desses, observou-se elementos quantitativos tais como o grau de interferência no desempenho das aplicações. Deve-se ressaltar que o presente trabalho é restrito à captura de mensagens; não foi posto como objetivo a classificação dessas informações e nem a adição de funcionalidades possíveis. Alguns resultados preliminares deste trabalho bem mais restritos do que os aqui contidos foram publicados anteriormente [FON 2001]. Uma síntese dos resultados globais será apresentada a seguir, neste capítulo.

O aspecto **transparência** foi constatado na implementação dos protótipos das diferentes abordagens. Também se identificou a existência de uma diferença quanto à forma como este aspecto é analisado: a transparência tanto pode ser vista como a capacidade de oferecer novas funcionalidades para a aplicação, como através da forma pela qual a aplicação fará uso dessas funcionalidades. Os mecanismos usados na implementação das abordagens de integração e interceptação permitem que a transparência seja facilmente alcançada, pois não há necessidade de qualquer conhecimento do uso do mecanismo por parte da aplicação. Isso ocorre de forma diferente na abordagem de serviço. Para que a aplicação se beneficie das funcionalidades oferecidas pelo mecanismo, ela deve referenciar explicitamente o serviço; logo, a funcionalidade é transparente, mas não o seu uso.

Outro aspecto que se pode verificar foi a **facilidade de uso**. A própria transparência oferecida por este mecanismo já traz consigo a facilidade de uso. Dentro da característica de cada mecanismo, eles proporcionam agilidade no desenvolvimento permitindo mais robustez e confiabilidade à aplicação, além de reduzir a possibilidade de erro no processo de implementação. As maiores diferenças entre os mecanismos

estão na análise da facilidade de uso se considerado o ponto de vista do programador do mecanismo, isto é, em sua própria implementação. Essa diferença pode ser ilustrada por elementos tais como: a documentação confusa relacionada à chamada *ptrace*, dificultando o seu uso; a boa documentação sobre como trabalhar com módulos; ou na falta de recursos quando se trabalha em nível de *kernel*, onde não é possível usar as bibliotecas disponíveis em nível de usuário, por exemplo; em contrapartida, na abordagem de serviço, pode-se contar com uma boa documentação e com vários recursos disponibilizados pelas linguagens.

No aspecto **portabilidade**, novamente ocorre uma similaridade entre os protótipos das abordagens de interceptação e integração, pois ambos se beneficiam de recursos de baixo nível oferecidos pelo ambiente, o sistema operacional, comprometendo assim, a sua portabilidade. Isso já não ocorre com a abordagem de serviço. Entretanto, o comprometimento potencial da portabilidade, nos enfoques de integração e interceptação, pode ser contornado. Para que a interceptação consiga prover este aspecto, tornando-se portável entre os demais sistemas operacionais baseados na plataforma *Unix*, pode utilizar-se de recursos da linguagem como as diretivas do pré-processador. Na integração, também o problema poderá ser contornado se houver características afins entre os sistemas. Por exemplo, se houver suporte à carga de módulos que possuam acesso a estrutura de dados das chamadas de sistema, é perfeitamente possível desenvolver um módulo que seja portável entre eles. Diferente das anteriores, a abordagem de serviço é independente do sistema operacional e construída sobre a aplicação, tornando-se portável.

O último aspecto avaliado, a **modularidade**, é um aspecto difícil de analisar porque está diretamente ligado com a forma como é implementada a funcionalidade. Para isso, usa-se a regra que componentes separados são mais modulares que os monolíticos. Seguindo a regra, pode-se afirmar que a modularidade é um aspecto presente em todas as abordagens. Para tanto, os diferentes mecanismos usados viabilizam que as funcionalidades implementadas com o seu uso sejam separadas da aplicação. Conseqüentemente, provêm componentes livres, que podem ter suas funcionalidades alteradas e incrementadas.

A análise quantitativa permitiu identificar o grau de interferência das diferentes abordagens no desempenho da aplicação. Foi realizado um número significativo de experimentos a fim de obter dados que permitissem uma avaliação estatística confiável. Os experimentos foram realizados em diferentes ocasiões, já que os primeiros resultados obtidos apresentaram números estranhos. A estratégia de implementação e os aspectos específicos de implementação foram revisados de tal forma que todos elementos que pudessem interferir sobre medidas, de forma indevida, fossem removidos. Superadas essas dificuldades iniciais, foram obtidos números apresentados no capítulo 5.

Analisando os experimentos, pode-se perceber que independentemente das características da aplicação, os mecanismos/abordagens seguem uma ordem crescente de interferência que se inicia pela abordagem de integração, onde foi usado um módulo para implementá-la; seguida da interceptação, implementada através da chamada de sistema *ptrace*; e por fim a de serviço, implementada através de um objeto. Os resultados demonstram, também, a pequena diferença numérica entre as abordagens de

integração e interceptação. A abordagem de serviço, por outro lado, causa uma perda de desempenho bastante significativa, chegando a duplicar o tempo da aplicação.

Existe uma grande quantidade de pesquisas no sentido de viabilizar projetos de sistemas operacionais que permitam explorar a capacidade de estender o sistema de forma mais fácil e eficiente. Como exemplos deste esforço podem ser citados os sistemas operacionais SPIN [BER 95] e VINO [SEL 94], entre outros sistemas que permitem a execução de código arbitrário dentro de seu espaço de *kernel*. Estes sistemas operacionais provêm um mecanismo perceptivo para extensões ao nível do usuário e expõem significativamente o estado intermo do *kernel* para as extensões.

Em contraste as estas pesquisas, na presente dissertação foram descritos alguns projetos que fazem uso dos benefícios da extensibilidade para adicionar novas funcionalidades aos sistemas. Esses benefícios podem ser explorados através de algumas características dos sistemas operacionais-padrão, como é o caso das interfaces de monitoramento disponibilizadas por sistemas como os baseados em Unix.

Dentro deste escopo, tem-se como idéia de trabalho futuro usar o conhecimento adquirido para desenvolver um módulo capaz de realizar a captura das chamadas de sistema. Este módulo deverá ter uma interface bem definida, para que seja possível a reutilização do mesmo. Sendo assim, ele poderá ser utilizado para os mais diversos fins, facilitando o acesso as informações contidas nas chamadas de sistema e possibilitando que essas sejam trabalhadas mais facilmente.

O módulo parece ser uma boa opção para o desenvolvimento desse tipo de mecanismo. Esta opção sustenta-se não só nos resultados quantitativos levantados nesse trabalho, os quais apontam o módulo como sendo o mecanismo com menos interferência, no que diz respeito à interposição de chamadas, mas também em facilidades de implementação. Estas facilidades são: a possibilidade de que o desenvolvimento das extensões ocorra tanto em nível do *kernel* como no do usuário; e a não obrigatoriedade de modificações no sistema operacional, nas aplicações existentes ou nas bibliotecas.

Anexo Implementações de Protótipos

A.1 – Código-fonte do protótipo de interceptação

```
// Interceptação - usa o attach
// Linux - kernel 2.2.14 - intercep.c
// Fonte Atz em: 06/04/2001

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <asm/user.h> //#include <sys/ptrace.h> já esta na user.h

# define LT_EV_NONE          0
# define LT_EV_SYSCALL       1
# define LT_EV_SYSRET        2
# define LT_EV_EXIT          3
# define LT_EV_EXIT_SIGNAL 4
# define LT_EV_SIGNAL        5

int main(){

struct user_regs_struct regs;
pid_t pid;
int status,s=0;

 int sig;
 int signum,sysnum_e,sysnum_s=-10;

 printf("\n ****  I N T E R C E P T A D O R   ****\n");
 printf("Informe o PID do processo a ser monitorado: ");
 scanf("%d",&pid);
 printf("\n< M E N S A G E N S >\n");

 if(ptrace(PTRACE_ATTACH,pid))
   printf("\nPTRACE_ATTACH - FALHA!\n");

while(1) {
  pid = wait(&status);
  if (pid == -1){
     printf("if do wait ...\n");
     if(errno==ECHILD){
        printf("No more children\n");
           break;
```

```
  } else if (errno==EINTR) {
    printf("wait received EINTR ?\n");
    sig = LT_EV_NONE;
    printf("LT_EV_NONE");
  }
  perror("wait");
  break;
}

if (WIFSTOPPED(status) && WSTOPSIG(status)==SIGTRAP){
   if(ptrace(PTRACE_GETREGS,pid,0,&regs)<0){
     printf("PT_GETREGS - Faut! \n");
   }else{
     sysnum_e= regs.orig_eax;
   }
   if (sysnum_e>=0) {
     if (sysnum_s != sysnum_e) { //entrada
        sysnum_s=sysnum_e;
        sig  = LT_EV_SYSCALL;
     } else {                  //saida
           sysnum_s=-10;
        sig = LT_EV_SYSRET;
     }
   }
} else {

     if (WIFEXITED(status)) {
        printf("WIFEXITED!\n");
        sig = LT_EV_EXIT;
        signum = WEXITSTATUS(status);
     }
     if (WSTOPSIG(status) != SIGTRAP) {
       printf("WSTOPSIG!\n");
          sig = LT_EV_SIGNAL;
       signum = WEXITSTATUS(status);
     }

 }
 // eventos do processo
 switch (sig) {
       case LT_EV_NONE:
          break;
       case LT_EV_SIGNAL: // Quando o sinal recebido não é um SIGTRAP
                printf("event: signal (%d)\n",signum);
       ptrace(PTRACE_SYSCALL,pid,1, signum);
                break;
       case LT_EV_EXIT:  // Sinal exit
                printf("event: exit (%d)",signum);
                break;
```

```
        case LT_EV_SYSCALL: // Trabalha c/ os dados na entrada da chamada
                printf("event: syscall (%d)\n",sysnum_e);
        if(sysnum_s==102) // Testa a chamada desejada (sockcall)
          s++;
        ptrace(PTRACE_SYSCALL,pid,1, 0);
                break;
        case LT_EV_SYSRET: // Trabalha c/ os dados na saida da chamada
                printf("event: sysret (%d)\n",sysnum_e);
        ptrace(PTRACE_SYSCALL,pid,1, 0);
                break;
        default:
                fprintf(stderr, "Error! unknown event?\n");
        exit(1);
   }

 }// while
 printf("Total de sockcall:%d\n",s);
 exit(0);
} //main
```

A.2 – Códio-fonte do protótipo de integração

```
// Integração
// Linux - kernel 2.2.14 - intrega.c
// Fonte Atz em: 02/08/2000

/* Padrões para módulo kernel*/
#include <linux/kernel.h>
#include <linux/module.h>
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <sys/syscall.h>  /* A lista das system calls */
#include <linux/sched.h>  /* para usar a struct task_struct, current->*/
#include <linux/uaccess.h>
#ifndef KERNEL_VERSION /*  versões do linux   */
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif

extern void *sys_call_table[]; /* A tabela das system call */

int our_pid; /*  definido para receber o parametro */
int ch[20];

MODULE_PARM(our_pid,"i");

/*Faz com que os parametros da função sejam passados todos em registradores*/
asmlinkage int (* originals_call)(int,unsigned long);

/* A "system call"  criada*/
asmlinkage int our_sys_socketcall(int call,unsigned long *args)
{
 if (our_pid == current->pid){
    ch[call]=ch[call]+1;
 }
 return originals_call(call,(long)args);
}

/* Initializa o modulo – retorna a system call */
int init_module()
{
 printk("INICIO DO MÓDULO INTI_MODULE\n");
 /* __NR_... é uma variavel calculada*/
 /*armazena a system call original */

 /*atribui as "system call criada" para a tabela */
 sys_call_table[__NR_socketcall] = our_sys_socketcall;
```

```
 return 0;
}

/* Cleanup – retira o registro do modulo do /proc */
void cleanup_module()
{
 int i,t=0;
 printk("FIM DO MÓDULO CLEANUP_MODULE\n");
 for (i=0;i<=19;i++){
   printk("Funcao - Numero de execucoes : %d  %d\n",i ,ch[i]);
   t=t+ch[i];
 }
 printk("Total de chamadas %d\n",t);

/*atribui as system calls originais a tabela */
  if (sys_call_table[__NR_socketcall]== our_sys_socketcall){
    sys_call_table[__NR_socketcall] = originals_call;
  }
 }
```

A.3 – Código-fonte do protótipo de serviço

A.3.1 – Código-fonte do protótipo Ping Pong

PingPongServer.java

```
/*
   Fazer um programa que recebe a string "ping" e escreve a string "pong" e continua
   alterando por 100 vezes; captura as mensagens através da abordagem de serviço.
*/

import java.net.*;
import java.io.*;

public class PingPongServer
{

  public static void main(String args[])
  {
    int cont=0;
    int porta = Integer.parseInt(args[0]);
    System.out.println("SERVER (proxy) INICIALIZADO!");
    try
    {

      ServerSocket ss=new ServerSocket(porta);

      while(true)
      {

        Socket s = ss.accept();

// Pega as streams de entrada e saída associadas ao Socket
        InputStream sIn = s.getInputStream();
        OutputStream sOut = s.getOutputStream();

// Cria um apontador para a escrita dos dados na stream de saída
        DataInputStream dis = new DataInputStream(sIn);
        DataOutputStream dos = new DataOutputStream(sOut);

        String strOut;
        String strIn;

        String ip=args[1];
        int porta_nova = Integer.parseInt(args[2]);
        Socket s_novo = new Socket(ip, porta_nova);

        // Pega as streams de entrada e saída associadas ao Socket
        InputStream sIn_novo = s_novo.getInputStream();
```

```
      OutputStream sOut_novo = s_novo.getOutputStream();

      // Cria apontadores para a leitura e escrita de dados das streams
      DataInputStream dis_novo = new DataInputStream(sIn_novo);
      DataOutputStream dos_novo = new DataOutputStream(sOut_novo);

      while(true)
      {
        strIn = new String(dis.readUTF()); // recebe do cliente
        //System.out.println("Recebeu do cliente: " + strIn);
        //cont++;
        //System.out.println("contador:"+cont);

        dos_novo.writeUTF(strIn); // envia para o serverserver
        strIn = new String(dis_novo.readUTF()); // recebe do serverserver
        //System.out.println("Recebeu do servidor: " + strIn);
        //System.out.println("Vai mandar para o cliente: " + strIn);

        dos.writeUTF(strIn);  // envia para o cliente
      }

      // Fecha os apontadors e streams associada ao socket
      //dos.close();
      //dis.close();
      //s.close();
    }
   }
   catch(IOException e)
   {
    //System.out.println(e);

    System.out.println("FIM DO SERVER! (proxy)");
    //System.out.println("Total do contador:"+cont);
   }
  }
}
```

PingPongServerServer.java

```
/*
  Fazer um programa que recebe a string "ping" e escreve a string "pong" e continua
  alterando por 100 vezes.
  -> A porta e o endereço ip devem ser passados pela linha de comando.
*/

import java.net.*;
import java.io.*;

public class PingPongServerServer
```

```
{

 public static void main(String args[])
 {

  int porta = Integer.parseInt(args[0]);
  System.out.println("SERVER INICIALIZADO!");
  try
  {

   ServerSocket ss=new ServerSocket(porta);

   while(true)
   {

    Socket s = ss.accept();

// Pega as streams de entrada e saída associadas ao Socket
    InputStream sIn = s.getInputStream();
    OutputStream sOut = s.getOutputStream();

// Cria um apontador para a escrita dos dados na da stream de saída
    DataInputStream dis = new DataInputStream(sIn);
    DataOutputStream dos = new DataOutputStream(sOut);

    String strOut;
    String strIn;

    while(true)
    {
     strIn = new String(dis.readUTF());
     System.out.println(strIn);
     if(strIn.equals("BYE"))
       break;
     else
     if(strIn.equals("Ping"))
       strOut = "Ping";
     else
       if(strIn.equals("Pong"))
         strOut = "Pong";
       else
         strOut = "erro";
     dos.writeUTF(strOut);
    }

// Fecha os apontadores e streams associadas ao socket
    dos.close();
    dis.close();
    s.close();
```

```
      }
    }
    catch(IOException e)
    {
      //System.out.println(e);
      System.out.println("FIM DO SERVER!");
    }
  }
}
```

PingPongClient.java

```
/*
   Fazer um programa que recebe a string "ping" e escreve a string "pong" e continua
   alterando por 100 vezes.
   -> A porta e o endereço ip devem ser passados pela linha de comando.
*/
import java.net.*;
import java.io.*;

public class PingPongClient
{
  public static void main(String args[]) throws IOException
  {
    String ip=args[0];
    int porta = Integer.parseInt(args[1]);

    Socket s = new Socket(ip, porta);

// Pega as streams de entrada e saída associadas ao Socket
    InputStream sIn = s.getInputStream();
    OutputStream sOut = s.getOutputStream();

// Cria apontadores para a leitura e escrita de dados das streams
    DataInputStream dis = new DataInputStream(sIn);
    DataOutputStream dos = new DataOutputStream(sOut);

    String strIn;
    String strOut = "Ping";
    long ti,tf;
    long t1,t2,r;
   //Tempo
    ti=System.currentTimeMillis();
    dos.writeUTF(strOut);
    for(int i=1;i<=100;i++)
    {
      strIn = new String(dis.readUTF());
      System.out.println(strIn);
      if(strIn.equals("BYE"))
```

```
        break;
       if(strIn.equals("Ping"))
        strOut = "Pong";
       else
        if(strIn.equals("Pong"))
          strOut = "Ping";
        else
          strOut = "erro";
       if(i==1000)
          strOut = "BYE";
       dos.writeUTF(strOut);
       for(int x=1; x<=2; x++)
       {
        t1=100;
        t2=2;
        r=0;
        r=((t1/t2)*t2);
        t1=((r*t2)/t2);
       }
     }
     tf=System.currentTimeMillis();
     System.out.println("\nTEMPO TOTAL:");
     System.out.println(tf-ti);

// Fecha todas as conexões e termina o aplicativo cliente
     dis.close();
     dos.close();
     System.out.println("FIM DO CLIENT!");
     s.close();
   }
}
```

A.3.2 – Código-fonte do protótipo Eleição de Líderes

Canal1.java

```
public class Canal1 extends Canal{

 private int msgEnviadas,msgRecebidas;
 private long ti,tf,tmptotal=0;

 public Canal1 (int numnodos, int porta){
  super(numnodos,porta);
 }

 public void enviaParaCliente(int id, int valor){
  msgEnviadas++;
  ti=System.currentTimeMillis();
  super.enviaParaCliente(id,valor);
  tf=System.currentTimeMillis();
  calculatempo(ti,tf);
 }

 public int recebeDeCliente(int num){
  int aux;
  msgRecebidas++;
  ti=System.currentTimeMillis();
  aux=super.recebeDeCliente(num);
  tf=System.currentTimeMillis();
  calculatempo(ti,tf);
  return aux;
 }
  private void calculatempo(long tmpi, long tmpf){
  tmptotal+=tmpf-tmpi;
 }
 public long tmptotal(){
 return(tmptotal);
 }
 public int munmsg(){
 return(msgEnviadas+msgRecebidas);
  }
}
```

Gerador.java

```
//Programa de PDP _ Gerador

import java.lang.*;
import java.io.*;
import java.net.*;
import java.util.*;
```

```
public class Gerador{
//  Essa Classe é a responsável por determinar qual é a topologia da
//rede em questão e quais os identificadores de cada processo que
//compõem a rede. O número de nodos da rede deve ser indicado no momento
//de início do programa (passado por parâmetro). Dado esse valor, a
//Classe Gerador cria uma matriz nxn randômica e booleana, contendo
//a informação de qual nodo está ligado com qual. Essa geração garante
//que não tenha nodos isolados. O Gerador, logo após, fica esperando
//uma conexão TCP de cada cliente (nodos da rede) até que seja preenchido
//o número de clientes especificado. Em seguida é passado para cada
//nodo da rede sua linha de topologia, bem como o número de nodos do
//sistema, seu identificador na Rede e o IP de todos os outros nodos
//da rede. Terminada essa passagem de informações o Gerador morre.

  private boolean topologia[][]; // contém a topologia da Rede (True = conectado)
  private Vector canais;         // contém os Sockets de conexão com cada
    //nodo da Rede
  private int numnodos;          // número de nodos da Rede

  public Gerador(int num){
  //  Construtor da Classe, apenas inicializa a variável numnodos
  //e chama os outros procedimentos inportantes da Classe.

    numnodos = num;
    this.geraTopologia();
    this.esperaNodos();
    this.enviaTopologia();
    System.out.println("#O processo lider, o Gerador, esta morrendo.");
    System.out.println("#Os Nodos devem escolher um novo lider.");
    System.out.println("#Para isso executam o algoritmo MaxFlood.");
    System.out.println();
    System.out.println("Por: Alexandre Cervieri & Joao Francisco H. Jornada");
  }

  private void geraTopologia(){
  //  Procedimento responsável pela criação da matriz da topologia.
  //Como dito acima, ele garante que não haverá nodos isolados.

    int i, j;
    topologia = new boolean[numnodos][numnodos];
    for (i=0;i<=(numnodos-1);i++){
      for (j=i;j<=(numnodos-1);j++){
        if (i == j){
          topologia[i][j] = false;
        }
        else{
          if ( Math.random()>= 0.6){ // Math.random() gera um número entre
                                     //0 e 1, e esse if determina uma taxa
```

```
                                //média de interconexão da rede de 30%
            topologia[i][j] = true;
            topologia[j][i] = true;
           }
           else{
            topologia[i][j] = false;
            topologia[j][i] = false;
           }
          }
         }
        }
        for (i=0;i<=(numnodos-1);i++){
         boolean todos = false;
         for (j=i;j<=(numnodos-1);j++){
          todos= topologia[i][j];
         }
         if (todos==false){          // De todos=False então o nodo está isolado
          //então o nodo vai ser ligado com o nodo a sua
           //direita (ou se for o último, vai ligar ao primeiro)
          if (i==(numnodos-1)){
           topologia[i][0] = true;
           topologia[0][i] = true;
          }
          else{
           topologia[i][i+1] = true;
           topologia[i+1][i] = true;
          }
         }
        }
        System.out.println();
        System.out.println("#Numero de nodos no sistema: " + numnodos);
        System.out.println();
        for (i=0;i<=(numnodos-1);i++){  // imprimindo a Topologia na Tela.
         for (j=0;j<=(numnodos-1);j++)
          if (topologia[i][j]==true) System.out.print(topologia[i][j] + "  ");
          else System.out.print(topologia[i][j] + " ");
         System.out.println();
        }
       }

       private void esperaNodos(){
       //  Procedimento que cria um ServerSocket (servidor) e espera conexões de todos
       //os nodos que compõem o sistema. Guarda os sockets criados em um vetor (Vector
canais),
       //o qual é criado (e inicializado) nesse mesmo procedimento.

        ServerSocket espera;
        Socket nodo;
        int i;
```

```
    try{
      canais = new Vector();
      espera = new ServerSocket(6015);  // cria o Servidor
      System.out.println();
      System.out.println("#Esperando conexao dos nodos em " +
InetAddress.getLocalHost().toString() + ":6015");
      System.out.println();
      for(i=0;i<=(numnodos-1);i++){
        nodo = espera.accept();    // servidor entra em accept, esperando conexões
          //assim que receber um pedido, será criado um
         //socket o qual é guardado no vetor canais.
        canais.addElement(nodo);
        System.out.println("#Nodo " + i + ". Localizacao: " +
nodo.getInetAddress().toString());
      }
    }
    catch(Exception e){
      System.out.println();
      System.out.println("Gerador.esperaNodos:" + e.toString());
      System.out.println();
      System.exit(0);
    }
  }

  public void enviaTopologia(){
  //  Já criadas todas as conexões com os nodos, este procedimento envia
  //para os nodos uma mensagem contendo as informações necessárias para
  //que os nodos possam, começar o algoritmo de Escolha de Líder.
  //O que é enviado é um buffer de bytes contendo as seguintes informações:
  // ------------------------------------------------------------------------
  // | n |top[id][0]...top[id][n] | id |-| ip[0] | / | ip[1] |...| / | ip[n] | - |
  // --|------------|----------------|---------------------|----------------------
  //   |            |                |->id do processo     |-> números IPs dos processos
  //   |            |->linha da topologia
  //   |-> número de nodos

    Socket nodo;
    OutputStream nodoOut;
    byte[] b;
    byte[][] top = new byte[numnodos][numnodos];
    byte[] linha = new byte[numnodos];
    byte[] ips = new byte[numnodos*4];
    int i, j, k, n, index;
    System.out.println();
    System.out.println("#Topologia: 0 indica nao ligado e 1 ligado.");
    System.out.println();
    for(i=0;i<=(numnodos-1);i++){
      for(j=0;j<=(numnodos-1);j++){
        if(topologia[i][j]==true) top[i][j] = 1;
```

```
    else  top[i][j] = 0;
    System.out.print(top[i][j] + " ");
   }
   System.out.println();
 }
 System.out.println();
 System.out.println("#Enviando linha de topologia para cada nodo.");
 System.out.println();
 Socket saux;
 index = 0;
 for(k=0;k<=(numnodos-1);k++){
   saux = (Socket) canais.elementAt(k);
   byte[] adr = saux.getInetAddress().getAddress();
   for(n=0;n<=(adr.length-1);n++){ //127 0 0 1
     ips[index] = adr[n];
     index++;
   }
 }
 try{
   for(i=0;i<=(numnodos-1);i++){
     for(j=0;j<=(numnodos-1);j++){
       linha[j] = top[i][j];
     }
     b = new byte[75];
     index = 0;

     // A partir daqui começa a montar a mensagem a ser enviada.

     b[index] = (byte) numnodos;  //numnodos
     index++;
     for(k=0;k<=(numnodos-1);k++){  //topologia[i][j]
       b[index] = linha[k];     // copia a linha da topologia
       index++;
     }
     b[index] = (byte) i;  //id
     index++;
     b[index] = 45;  //-
     index++;
     for(k=0;k<=(numnodos-1);k++){
       for(n=0;n<=3;n++){ //127 0 0 1
         b[index] = ips[n+(4*k)];     // copia os endereços IP byte a byte
         index++;
       }
       b[index] = 47;  // "/"
       index++;
     }
     b[index-1] = 45; //-
     nodo = (Socket) canais.firstElement();    // pega o socket do Vetor
```

```
        System.out.println(nodo.getInetAddress().getHostAddress() + " " +
nodo.getPort());
        nodoOut = nodo.getOutputStream();
        nodoOut.write(b);        // envia a mensagem
        System.out.println("Enviou dados para: " + i);
        for(k=0;k<=(index-1);k++)
          System.out.print(b[k] + " ");
        System.out.println();
        System.out.println();
        nodo.close();     // encerra o socket
        canais.removeElement(nodo);   // remove o objeto do vetor
      }
    }
    catch(Exception e){
      System.out.println();
      System.out.println("Gerador.enviaTopologia:" + e.toString());
      System.out.println();
      System.exit(0);
    }
  }

  public static void main(String[] args){
    Gerador gerador;
    int num;
    if(args.length!=1){
      System.out.println();
      System.out.println("Deve ser determinado o numero de nodos.");
      System.out.println();
      System.exit(0);
    }
    else{
      Integer i = new Integer(args[0]);
      num = i.intValue();
      if (num>12) num = 12;     // Esse limite de nodos foi usado apenas para
                                //fins de visualização na tela, se não,
      //não há limite.
      gerador = new Gerador(num);  // Inicia a classe Gerador.
    }
  }
}
```

Arquivo Nodos.java

```
//Programa de PDP _ Nodos

import java.lang.*;
import java.io.*;
import java.net.*;
import java.util.*;
```

```
import Canal;

public class Nodos{
//  Classe respnsável por implementar os processos clientes do Gerador.
//É nessa classe que irá ser executado o algoritmo MaxFlood realmente.
//Várias instâncias dessa classe devem ser criadas, em uma mesma máquina
//ou em máquinas distintas, até que o Gerador receba o número de
//pedidos de conexões que ele esteja esperando (número de nodos do
//sistema passado como parâmetro). Essa classe cria uma instância da
//classe Canal que irá se repnsabilizar pela comunicação com os nodos
//vizinhos.

  private InetAddress gerador;
  private byte linha[];
  private int numnodos;
  private int numvizinhos;
  private int ID;
  private Vector str_ip;
  private Canal1 canal;
  private long ti,tf;
  public Nodos(InetAddress gera){
  //  Construtor da classe, apenas chama as rotinas importantes da classe
  //e depois finaliza o processo.

    //Teste dar tempo de ver o PID do processo
    System.out.println("Antes do sleep!!!");
    try
    {
       Thread.sleep(8000);
    } catch(Exception e){}
    System.out.println("Depois do sleep!!!");

    gerador = gera;
    conectaGerador();
    conectaVizinhos();
    executaMaxFlood();
    System.exit(0);
  }

  private void conectaGerador(){
  //  Função responsável por efetuar a conexão com o servidor e receber a
  //mensagem contendo informações da rede. Além de receber a mensagem, é de
  //sua responsabilidade decodificar (ou seja, pegar os bytes e transformar
  //nos tipos de dados que interessam para o programa) a mensagem.

    Socket canal;  // canal com o gerador
    InputStream canalIn;  // stream de entrada que irá receber a mensagem do Gerador
    byte[] b = new byte[75];  // buffer de recebimento (tam máximo para até 12 nodos na
rede)
```

```
    int i, j, index;
    int i_ip3, i_ip2, i_ip1, i_ip0;
    try{

      System.out.println("inicio da Conexao -> tempo medio");
      ti=System.currentTimeMillis();

      canal = new Socket( gerador, 6015);  // Tenta se conectar com o Gerador em
host:6000
      System.out.println();
      System.out.println("# Gerador: " + canal.getInetAddress().toString() + ":" +
canal.getPort());
      System.out.println();
      canalIn = canal.getInputStream();  // já conctado, obtém o stream de recebimento
      canalIn.read(b);  // fica bloqueado até que o Gerador envie a mensagem
      canal.close();     // fecha o canal de comunicação com o Gerador
      for(i=0;i<=(b.length-1);i++){
        System.out.print(b[i] + " "); //imprime na tela o que recebeu
      }
      System.out.println();
      System.out.println();

      // Começa a decodificar a mensagem

      index = 0;
      numnodos = (int) b[index]; // número de nodos do sistema
      index++;
      System.out.println("#Numero de nodos do sistema: " + numnodos);
      System.out.println();
      System.out.println("#Linha da topologia:");
      linha = new byte[numnodos];
      for(i=0;i<=(numnodos-1);i++){
        linha[i] = b[index]; //obtém a linha da topologia do buffer recebido
        index++;
        System.out.print(linha[i] + " ");
      }
      System.out.println();
      System.out.println();
      ID = (int) b[index];  // obtém o ID que o Gerador atribuiu para esse processo
      index++;
      System.out.println("#Indentificador do Processo: " + ID);
      System.out.println();
      index++; //Pula o caracter de controle -
      System.out.println("#Preparando para pegar IPs.");
      System.out.println();
      str_ip = new Vector(); // vetor que guardará os IPs dos nodos da rede (todos)
      for(i=0;i<=(numnodos-1);i++){
        i_ip3 = b[index]; index++;
        i_ip2 = b[index]; index++;
```

```
        i_ip1 = b[index]; index++;
        i_ip0 = b[index]; index++;
        index++; //pula caracter de controle "/"

 // Como os IPs podem chegar como bytes negativos, eles são convertidos para positivos
        if (i_ip3 < 0) i_ip3 += 256;
        if (i_ip2 < 0) i_ip2 += 256;
        if (i_ip1 < 0) i_ip1 += 256;
        if (i_ip0 < 0) i_ip0 += 256;

        // Monta uma string com os bytes recebidos para remontar o endereço IP
        String ip = new String(i_ip3+"."+i_ip2+"."+i_ip1+"."+i_ip0);
        System.out.println("Nodo :" + i + " tem IP = " + ip);
        str_ip.addElement(ip);  //adiciona o IP no vetor
      }
     }
     catch(IOException e){
      System.out.println();
      System.out.println("Nodos.conectaGerador:" + e.toString());
      System.out.println("Ocorreu exceção nao computar esse tempo ...");
      System.out.println();
      System.exit(0);
     }
    }

    private void conectaVizinhos(){
    //  Procedimento responsável por criar os objetos que farão a comunicação
    //com os nodos vizinhos.

     int i, numviz = 0;
     for(i=0;i<=(numnodos-1);i++){
      if( linha[i]==1) numviz++;
     }
     numvizinhos = numviz;
     System.out.println();
     System.out.println("#Criando canal com numvizinhos= " + numviz + " Porta
servidor: " + (7000+ID));
     canal = new Canal1( numviz, 7000 + ID);//cria objeto canal que irá criar
//um servidor (ServerSocket)
    for(i=0;i<=(numnodos-1);i++){
      if( linha[i]==1){
        String sip = (String) str_ip.elementAt(i);
        canal.criaCliente( sip, i);//cria cliente i, com servidor 700+id
      }
     }
     // espera conexoes do servidor
     System.out.println("#Espera que todos vizinhos se conectem.");
     while (!canal.pronto()) {}
     System.out.println("#Vizinhos conectados, vai comecar o MaxFlood!");
```

```
  }

  private void executaMaxFlood(){
  //  Esse procedimento é responsável pela implementação do algoritmo
  //MaxFlood. Aqui o código fica muito semelhante àquele apresentado
  //por Linch, com uma simplificação; o número de rodadas para nós
  //não é igual ao diâmetro da rede, mas sim ao número de nodos da rede.

    int max_id = ID;  // o id máximo no início é igual ao próprio ID do processo
    int id = 0;       // variável para recebimento de id dos outros processos
    String status;    // status do processo (LÍDER ou NÃO-LÍDER)
    int rounds = 0;   // indica o número do round atual
    int i;
    System.out.println("# Iteracoes esperadas: "+numnodos);
    for(rounds=0;rounds<numnodos;rounds++){  // laço do algoritmo
      System.out.println("#Rodada: "+ rounds);
      for(i=0;i<numvizinhos;i++){  // envia o seu max_id para todos os vizinhos
        canal.enviaParaCliente(i,max_id);
        System.out.print("E" + i + ":" + max_id + " ");
      }
      System.out.println();
      for(i=0;i<numvizinhos;i++){  // recebe o max_id de cada vizinho
        id = canal.recebeDeCliente(i);
        System.out.print("R" + i + ":" + id + " ");
        if (max_id<id) max_id = id;  //  compara o max_id recebido com o seu
         //se for maior que o seu, atualiza o seu,
         //senão for maior, permanece o mesmo valor.
      }
      System.out.println();
    }
    try{
      // No final se o max_id for o seu, ele é o líder, senão ele é não-líder
      if (max_id==ID) status =  new String("--->>>SOU LIDER<<<---, Meu ID = " + ID
+ " Meu IP = " + InetAddress.getLocalHost());
      else status = new String("<NAO SOU LIDER> Meu ID="+ID+" Meu
IP="+InetAddress.getLocalHost()+" ID do LIDER = "+max_id);

      tf=System.currentTimeMillis();
      System.out.println("\n*** TEMPO TOTAL:");
      //System.out.println(tf-ti);

      System.out.println(canal.tmptotal());
      System.out.println("\nNumero de msg:");
      System.out.println(canal.munmsg());
      System.out.println("fim da conexao -> tempo medido");

      System.out.println();
      System.out.println(status);
      System.out.println();
```

```
      System.out.println();
      System.out.println("Por: Alexandre Cervieri & Joao F. H. Jornada");
    }
    catch(Exception e){
      System.out.println();
      System.out.println("Nodos.executaMaxFlood:" + e.toString());
      System.out.println();
    }
  }

  public static void main(String[] args){
    //  Para poder inicializar o objeto Nodos, é necessário passar o
    //IP do processo Gerador.

    Nodos nodoi;
    if (args.length!=1){
      System.out.println();
      System.out.println("Deve ser informado o IP(HostName) do Gerador.");
      System.out.println();
      System.exit(0);
    }
    else{
      try{
        InetAddress addr = InetAddress.getByName(args[0]);
        nodoi = new Nodos(addr);
      }
      catch(Exception e){
        System.out.println();
        System.out.println("Nodos.main:" + e.toString());
        System.out.println();
        System.exit(0);
      }
    }
  }
}
```

Arquivo Canal.java

```
//Programa de PDP _ Canal

import java.lang.*;
import java.net.*;
import java.io.*;
import java.util.*;

class Servidor extends Thread{
//  Classe reponsável por criar um servidor ( em paralelo com a execução normal
//do programa, visto que essa classe é derivada de Thread), para que possam ser
//aceitas as conexões dos vizinhos do nodo. A criação de um canal bidirecional,
```

```
//portanto, entre os nodos vizinhos, é feita através da criação de duas conexões
//TCP. Cada nodo cria sua classe Servidor que ficará esperando por conexões,
//e depois cada nodo irá tentar criar conexões com os outros servidores.
//  O servidor ainda é responsável por receber dados dos canais TCP.

  private InetAddress endereco; //endereco local
  private int porta;  //porta local igual sempre a 7000 + id
  private int numnodos;  // número de nodos vizinhos que ele deverá esperar conexão
  private ServerSocket socketWait;
  private Vector conexoes;
  public boolean conectados;

  public Servidor(int num, int port){
  // Construtor da classel, apenas inicializa variáveis.

    try{
      endereco = InetAddress.getLocalHost();
      porta = port;
      numnodos = num;
      conectados = false;
    }
    catch(UnknownHostException e){
      System.out.println();
      System.out.println("Servidor.Servidor:" + e.toString());
      System.out.println();
    }
  }

  public void run(){
  //  Função que é executada quando se faz um start em uma Thread. Essa
  //função que é a mais importante do servidor. Ela cria um ServerSocket
  //para esperar as conexões dos vizinhos, aceitas até o número de vizinhos
  //de conexões e depois fica em um laço ilimitado esperando requisição
  //de leitura de valor.

    Socket servidor;
    int i;
    try{
       conexoes = new Vector();
       socketWait = new ServerSocket(porta);
       System.out.println();
       System.out.println("#Criado servidor em: " + InetAddress.getLocalHost() + ":" +
porta);
       for(i=0;i<numnodos;i++){  // aceita conexões até o número de vizinhos
         servidor = socketWait.accept();
         System.out.println("#IP: " + servidor.getInetAddress().toString() + ":" +
servidor.getPort() + " conectado.");
         conexoes.addElement(servidor);
       }
```

```
      System.out.println("#Servidor recebeu todas as Conexões.");
      conectados = true;
    }
    catch(IOException e){
      System.out.println();
      System.out.println("Servidor.run:" + e.toString());
      System.out.println();
    }
    while (true) {}
  }

  public int recebe(int num){
  //  Procedimento que lê um byte do canal de comunicação especificado por
  //num. Fica bloqueado até que receba um byte.

    InputStream servidorIN;
    Socket cliente;
    byte[] b = new byte[1];
    b[0] = -1;
    int i=0;
    try{
      cliente = (Socket) conexoes.elementAt(num); // obtém o Socket da conexão num
      servidorIN = cliente.getInputStream();
      servidorIN.read(b);  // fica bloqueado até que tenha um byte para ler
      return b[0];  //retorna apenas o primeiro byte (o que importa) do buffer
    }
    catch(IOException e){
      System.out.println();
      System.out.println("Servidor.recebe:"+e.toString());
      System.out.println();
      return -1;
    }
  }
}

class Cliente {
  private InetAddress endereco;
  private int porta;
  private Socket cliente;
  private int ID;
  private boolean conectado;

  public Cliente(String ip, int id){
    try{
      endereco = InetAddress.getByName(ip);
      porta = id + 7000;
      ID = id;
      conectado = false;
      cliente = null;
```

```
    }
    catch(UnknownHostException e){
      System.out.println();
      System.out.println("Cliente.Cliente:" + e.toString());
      System.out.println();
    }
  }

  public void run(){
    while(cliente == null){
      try{
        cliente = new Socket( endereco, porta);
        System.out.println("#Cliente de: " + cliente.getInetAddress().toString() + ":" +
cliente.getPort());
        conectado = true;
      }
      catch(Exception e){
        System.out.println();
        System.out.println("Cliente.run:" + e.toString());
        System.out.println();
      }
    }
  }

  public boolean getConectado(){
    return conectado;
  }

  public void envia(int valor){
    OutputStream clienteOUT;
    byte val = (byte) valor;
    try{
      clienteOUT = cliente.getOutputStream();
      clienteOUT.write(val);
      //System.out.println("#Enviou valor.");
    }
    catch(IOException e){
      System.out.println();
      System.out.println("Cliente.envia:" + e.toString());
      System.out.println();
    }
  }
}

public class Canal{
//  Classe que encapsula os objetos e dados necessários para a comunicação
//de um nodo (classe Nodos) com os seus vizinhos. É responsável por deixar
//mais simples para a Classe Nodos funções como estabelecimento de conexão
//com os vizinhos e recebimento e envio de dados.
```

```
  private Servidor servidor;
  private Vector clientes;
  private int numviz;
  public boolean pronto(){return(servidor.conectados);}

  public Canal(int numnodos, int porta){
  //  Construtor da classe, além de inicializar variáveis, cria um servidor
  //que espera conexões dos vizinhos do processo.

    numviz = 0;
    servidor = new Servidor(numnodos,porta);
    servidor.start();
    clientes = new Vector();
  }

  public void criaCliente( String str_ip, int viz_id){
  //  Procedimento que cria uma conexão com algum dos vizinhos e adiciona
  //o objeto a lista de conexões com vizinhos atuais.

    Cliente cliente = new Cliente(str_ip,viz_id);
    cliente.run();
    clientes.addElement(cliente);
    numviz++;
  }

  public void enviaParaCliente( int id, int valor){
  //  Procedimento que envia um byte(valor) para o canal de comunicação
  //identificado por id.

    Cliente cliente;
    cliente = (Cliente) clientes.elementAt(id);
    cliente.envia(valor);
  }

  public int recebeDeCliente(int num){
  //  Procedimento que le algum byte de um canal de comunicação.
      return(servidor.recebe(num));
  }
}
```

Bibliografia

[ALE 97] ALEXANDROV, A. D. et al. Extending the Operating System at the User Level: the Ufo Global File System. In: ANNUAL TECHNICAL CONFERENCE ON UNIX AND ADVANCED COMPUTING SYSTEMS, 1997, Anaheim, CA. **Proceedings...** Boston, MA: USENIX, 1997.

[ALE 98a] ALEXANDROV, A. D.; MAXIMILIAN, I.; SCHAUSER K. E. Ufo: a Personal Global File System Based on User-Level Extensions the Operating System. **ACM Transactions on Computer Systems**, New York, v.16, n.3, p.207-233, Aug.1998.

[ALE 98b] ALEXANDROV, A. D.; KMIEC, P.; SCHAUSER, K. **Consh:** Confined Execution Environment for Internet Computations. Santa Barbara, CA: University of California, 1998. Disponível em: <http://www.cs.ucsb.edu/~berto/ papers/index.html>. Acesso em: 18 jul. 2001.

[ALE 99] ALEXANDROV, A. D. **User-Level Operating System Extensions Based on System Call Interposition.** Santa Barbara, CA: University of California, 1999. Disponível em: <berto@cs.ucsb.edu>. Acesso em: 25 jul. 2001.

[BAB 97] BABAOGLU, O.; BATOLI, A. Selecting a "Primary Partition" in Partitionable Asynchronous Distributed Systems. In: IEEE SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, 1997, Durham, NC. **Proceedings…** New York: IEEE Computer Society Press, 1997.

[BAB 99] BABAOGLU, O.; DAVOLI, R.; MONTRESOR, A. **Group Communication in Partitionable Systems:** Specification and Algorithms. [S.l]: UBLCS, 1999. Disponível em: <http://www.cs.unibo.it/projects/jgroup/publications.htnl>. Acesso em: 15 maio. 2000.

[BER 95] BERSHAD, B. et al. Extensibility, Safety and Performance in the SPIN Operating System. In: ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 15., 1995, Copper Mountain Resort, CO. **Proceedings…** New York: ACM, 1995

[BER 98] BERCK, M.et al. **Linux *Kernel* Internals**. Harlow: Addison Wesley, 1998.

[BIR 96] BIRMAN, K.P. **Building Secure and Reliable Network Applications.** Greenwich: Manning Publications Co., 1996. 591 p.

[BRZ 95] BRZEZINSKY, J.; HELARY J.-M.; RAYNAL M. **Semantic of**

Recovery Lines for Backward Recovery in Distributed Systems. [S.l.]: Institut National de Recherche en Informatique et en Automatique, 1995. (Rapport de Recherche, RR-2468)

[CHI 93] CHIBA, S.; MASUDA, T. Designing an Extensible Distributed Language with a Meta-Level Architecture. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, ECOOP, 7., 1993, Kaiserlautern. **Object-oriented programming.** Berlin: Springer-Verlag, 1993. p. 482-501. (Lecture notes in Computer Science, v.707).

[CHI 95] CHIBA, S. A Metaobject Protocol for C++. **SIGPLAN Notices**, New York, v.30, n.10, p.285-299, Oct. 1995. Trabalho apresentado na 10. conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA, 1995.

[CUK 98] CUKIER, M. et al. AQUA: An Adaptive Architecture that Provides Dependable Distributed Objects. In: IEEE SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEM, 3., 1998, [S.l.]. **Proceedings…** New York: IEEE Computer Society Press, 1998

[CUK 99] CUKIER, M. et al. Building Dependable Applications Using AQUA. In: IEEE SYMPOSIUM ON HIGH ASSURANCE SYSTEMS ENGINEERING, 4., 1999, Washington, D.C. **Proceedings…** New York: IEEE Computer Society Press, 1999.

[DEF 98] DÉFAGO, X. et al. Semi-Passive Replication. In: IEEE SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, SRDS, 1998, West Lafayette. **Proceedings…** New York: IEEE Computer Society Press, 1998.

[FAB 96] FABRE, J.C. et al. FRIENDS: A Flexible Architecture for Implementing Fault Tolerant and Secure Distributed Applications. In: EUROPEAN DEPENDABLE COMPUTING CONFERENCE, EDCC-2, 1996, [S.l.]. **Proceedings…** Berlin: Springer Verlag, 1996.

[FAB 98] FABRE, J.C.; Pérennou, T. A Metaobject Architecture for Fault-Tolerant Distributed Systems. **IEEE Transactions on Computers**, New York, v.47, n.1, p. 78-95, Jan. 1998.

[FEL 96] FELBER, P. et al. The Design of a CORBA Group Communication Service. In: IEEE SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, SRDS-15, 1996, Niagara-on-the-Lake, Canada. **Proceedings…** New York: IEEE Computer Society Press, Oct.1996.

[FEL 97a] FELBER, P.; GUERRAOUI, R.; SCHIPER, A. **A CORBA Object Group Service**. [S.l.]: École Polytechnique Fédérale de Lausanne, Switzerland, 1997. Disponível em: <http://lsewww.epfl.ch/~rachid/papers/ publis97.html>. Acesso em: 10

jul. 2000.

[FEL 97a] FELBER, P.; GARBINATO, B.; GUERRAOUI, R. **Towards Reliable CORBA Integration vs. Service Approach.** In: MÜHLHÄUSER, Max. Special Issues in Object-Oriented Programming. [S.l.]: dpunkt-Verlag, 1997. p. 199-205

[FEL 98a] FELBER, P. **The CORBA Object Group Service: A Service Approach to Object Groups in CORBA**. 1998. Ph.D. Thesis - École Polytechnique de Lausanne. Disponível em: <http://lsewww.epfl.ch/~felber>. Acesso em: 10 mar. 2000.

[FEL 98b] FELBER, P.; GUERRAOUI, R.; SCHIPER, A. The Implementation of a CORBA Object Group Service. **Theory and Practice of Object Systems**, New York, v.4, n.2, p. 93-105, Apr. 1998.

[FEL 98c] FELBER, P. et al. Evaluating CORBA Portability: The Case of an Object Group Service. In: INTERNATIONAL ENTERPRISE DISTRIBUTED OBJECT COMPUTING WORKSHOP, 1998, San Diego, CA. **Proceedings…** [S.l.: s.n.], 1998.

[FEL 98d] FELBER, P.; GUERRAOUI, R.; WIESMAM, M. **Programming with Object in CORBA**. Lausanne: École Polytechnique de Lausanne, 1988. Disponível em: <http://lsewww.epfl.ch/~felber>. Acesso em: 15 jun. 2000.

[FEL 99] FELBER, P. et al. Replicating Objects using the CORBA Event Service. In: IEEE COMPUTER SOCIETY WORKSHOP ON FUTURE TRENDS IN DISTRIBUTED COMPUTING SYSTEMS, FTDCS-6, 1997, Tunis, Tunisia. **Proceedings…** New York: IEEE Computer Society Press, 1997.

[FON 2001] FONTOURA, A. B. et al. Evaluating approaches of the capturing of application information. In: IEEE LATIN-AMERICAN TEST WORKSHOP, 2001, Cancun, Mexico. **Proceedings…** New York: IEEE Computer Society Press, 2001.

[GHO 98] GHORMLEY, D. P. et al. SLIC: An Extensibility System for Commodity Operating Systems. In: USENIX TECHNICAL CONFERENCE, 1998, New Orleans, Louisiana. **Proceedings…** Berkeley, CA: USENIX, 1998.

[GOL 96] GOLDBERG, I. et al. A Secure Environment for Entrusted Helper Applications: Confining the Wily Hacker. In: USENIX UNIX SECURITY SYMPOSIUM, 1996, San Jose, California. **Proceedings…** Berkeley, CA: USENIX, 1996.

[HAD 93] HADZILACOS, V.; TOUEG, S**. Fault-Tolerant Broadcasts and Related Problems**. [S.l.]: Addison-Wesley, 1993.

[HUA 93] HUANG, Y; KINTALA, C. Software Implemented Fault Tolerance: Technologies and Experience. In: FAULT-TOLERANT COMPUTING SYSTEMS SYMPOSIUM, 23., 1993, [S.l.]. **Proceedings…** [S.l.: s.n.], 1993.

[JAI 91] JAIN, R. **The Art of Computer Systems Performance Analysis:** Techniques for Experimental Design, Measurement, Simulation, and Modeling. New York: John Wiley, 1991

[JAN 2000a] JAIN, K., SEKAR, R. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In: NETWORK AND DISTRIBUTED SYSTEMS SYMPOSIUM, 2000, San Diego, CA. **Proceedings…** [S.l.: s.n.], 2000.

[LUN 2000a] LUNG, L C.; JONI, S.F.; JEAN-MARIE, F.; JORGE, R.S.O. Experiências com Comunicação de Grupo nas Especificações Fault-Tolerant CORBA. In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES, 18., 2000, Belo Horizonte. **Anais…** Belo Horizonte: UFMG, 2000.

[LUN 2000b] LUNG, L C. et al. GrouPac: Um Framework para implementação de Aplicações Tolerantes a Falhas. In: CONFERÊNCIA LATINO AMERICANA DE INFORMÁTICA, CLEI, 26., 2000, Cidade do México, México. **Memórias.** México: Tec de Monterrey, 2000.

[LUN 97] LUNG, L C. et al. Uma Abordagem Reflexiva Usando Suporte de Grupo para Implementar Técnicas de Replicação em Ambientes Heterogêneos. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, SCTF´97, 7., 1997, Campina Grande, PB. **Proceedings…** Porto Alegre: Institutu de Informática da UFRGS, 1997.

[LAN 97] LANDIS, S.; MAFFEIS, S. Building Reliable Distributed Systems with CORBA. **Theory and Practice of Object Systems**, New York, v. 3, n. 1, p. 31-43, Apr. 1997.

[LUN 99a] LUNG, L C. et al. CosNoming FT - Fault-Tolerant CORBA Naming Service. In: IEEE SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEM, 1999, Lausanne, Switzerland. **Proceedings…** New York: IEEE Computer Society Press, 1999.

[LUN 99b] LUNG, L C. et al. MetaFT - A Reflective Approach to Implement Replication Techniques in CORBA. In: INTERNATIONAL CONFERENCE OF THE CHILEAN COMPUTER SCIENCE SOCIETY, SCCC, 1999, Talca, Chile. **Proceedings…** New York: IEEE Computer Society Press, 1999.

[LYN 96] LYNCH, Nany A. **Distributed Algorithms.** San Francisco: Morgan Kaufmann, 1996. p.52-57.

[MAF 95a] MAFFEIS, S. **Run-Time Support for Object-Oriented Distributed Programming.**, 1995. Ph.D. Thesis - University of Zurich, Zurich. Disponível em: <http://www.softwired-inc.com/people/maffeis/electra/electra_thesis>. Acesso em: 06 maio 2000.

[MAF 95b] MAFFEIS, S. Adding Group Communication and Fault-Tolerance to CORBA, In: USENIX CONFERENCE ON OBJECT-ORIENTED TECHNOLOGIES, 1995, Monterey, CA. **Proceedings…** Berkeley, CA: USENIX, 1995.

[MAF 96] MAFFEIS, S. The Object Group Desing Pattern. In: USENIX CONFERENCE ON OBJECT-ORIENTED TECHNOLOGIES, 1996, Toronto, Camada. **Proceedings…** [S.l.:s.n.], 1996.

[MAX 2000] MAXWELL, S. ***Kernel* do Linux**. São Paulo: Makron Books, 2000.

[MIT 2000] MITCHEM, T. et al. Using *Kernel* Hypervisors to Secure Applications, In: ANNUAL COMPUTER SECURITY APPLICATION CONFERENCE, 1997, [S.l.]. **Proceedings…** [S.l.:s.n.], 2000.

[MON 99b] MONTRESOR, A. A Reliable Registry for the Jgroup Distributed Object Model. In: EUROPEAN RESEARCH SEMINAR ON ADVANCES IN DISTRIBUTED SYSTEMS, 1999, Madeira, Portugal. **Proceedings…** [S.l.:s.n.], 1999.

[MON 99c] MONTRESOR, A. et al. **Group-Enhanced Remote Method Invocations.** [S.l.]: UBLCS, 1999. Technical Report. Disponível em: http://www.cs.unibo.it/projects/jgroup/publications.html. Acesso em: 03 jan. 2000.

[MON 99d] MONTRESOR, A. The Jgroup Reliable Distributed Object Model. In: INTERNATIONAL WORKING CONFERENCE ON DISTRIBUTED APPLICATIONS AND INTEROPERABLE SYSTEMS, 1999, Helsinki, Finland. **Proceedings…** [S.l.:s.n.], 1999.

[MON99a] MONTRESOR, A.; DAVOLI, R.; BABAOGLU, O. **Middleware Suport for the Development of Distributed Application in Partitionable Systems**. [S.l.]: UBLCS, 1999. Technical Report. Disponível em: <http://www.cs.unibo.it/projects/jgroup/publications.html>. Acesso em: 06 jan. 2000.

[MOS 98] MOSER, L.E.; MELLIAR-SMITH, P. M.; NARASIMHAN, P. Consistent Object Replication in the Eternal System. **Theory and Practice of Object Systems**, New York, v.4, n.2, p. 81-92, Apr. 1998.

[NAR 97a] NARASIMHAN, P. et al. Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Tolerance. In: USENIX

CONFERENCE ON OBJECT-ORIENTED TECHNOLOGIES AND SYSTEMS, 1997, Portland, Oregon. **Proceedings…** [S.l.:s.n.], 1997.

[NAR 97b] NARASIMHAN, P.; MOSER, L.E.; MELLIAR-SMITH, P.M. The Intercetion Approach to Reliable Distributed CORBA Objects. In: USENIX CONFERENCE ON OBJECT-ORIENTED TECHNOLOGIES AND SYSTEMS, 1997, Portland, Oregon. **Proceedings…** [S.l.:s.n.], 1997.

[NAR 99] NARASIMHAN, P. et al. Using Interceptors to Enhance CORBA. **Computer**, New York, v.32, n.7, p.62-68, July 1999.

[PET 95] PETRI, S.; LANGENDÖRFER, H. Load Balancing and Fault Tolerance in Workstation Clusters Migrating Groups of Communicating Processes. **Operating Systems Review**, New York, v. 29, n. 4, p.25-36, Oct. 1995.

[PET 98a] PETRI, S.; BOLZ, M.; LANGENDÖRFER, H. Migration and Rollback Transparency for Arbitrary Distributed Applications in Workstation Clusters. In: WORKSHOP ON PARALLEL AND DISTRIBUTED PROCESSING, 1998, Orlando, Florida. **Proceedings…** [S.l.:s.n.], 1998.

[PET 98b] PETRI, S.; BOLZ, M.; LANGENDÖRFER, H**. Transparent Migration and Rollback for Arbitrary Distributed Applications in Workstation Clusters.** Germany: Braunschweig University of Technology, 1998.

[POM 99] POMERANTZ, O. **Linux Kernel Module Programming Guide**. [S.l.: s.n.], 1999. Disponível em: <http://metalab.unc.edu/pub/linux/docs/linux-doc-projectmodule-programming-guie>. Acesso em: 16 mar. 2000.

[RIC 97] RICCIARDI. A.; OGG, M.; PREVIATO, F. **Experience with Distributed Replicated Objects**: The Nile Project. [S.l.:s.n.], 1997. Disponível em: <http:// maple.ece.utexas.edu/>. Acesso em 21 abr. 2000.

[RUB 99] RUINI, A. **Linux Device Drivers.** São Paulo: Market Books, 1999.

[RUS 97] RUSHIKESH, K.; VIVEKANANDA, N.; RAM, J. Message Filter for Object-oriented Systems. **Software - Practice and Experience**, London, v.27 n.6, p.677-699, June 1997.

[SAB 99] SABNIS, C. et al. Proteus: A Flexible Infrastructure to Implement Adaptive Fault Tolerance. In: IFIP INTERNATIONAL WORKING ON DEPENDABLE COMPUTING FOR CRITICAL APPLICATIONS, 7., 1997, San Jose, CA. **Proceedings…** [S.l.:s.n.], 1999.

[SCH 90] SCHILDT, H. **C Completo e Total.** São Paulo: McGraw-Hill, 1990. p.269-275.

[SEL 94] SELTZER, M. et al**. An Introduction to the Architeture of the VINO Kernel**. Harvard: Harvard University, 1994. (Technical Report TR34-94). Disponível em: <http://www.eecs.harvard.edu/~margo/papers/>. Acesso em: 21 nov. 1999.

[TAN 92] TANENBAUM, A.S. **Sistemas Operacionais Modernos.** São Paulo: Prentice Hall do Brasil, 1992. p.304-311.

[VAH 96] VAHALIA.U. **Unix Internals:** The New Frontiers. Upper Saddle River, NJ: Prentice Hall, 1996.

[WAG 99] WAGNER, D.A. **Janus:** an Approach for Confinement of Entrusted Applications**.** 1999. Master's thesis - University California, Berkeley. Disponível em: <http://www.cs.berkeley.edu/~daw/papers>. Acesso em: 21 out. 1999.