Universidade Federal de Goiás
Instituto de Informática

Jesus José de Oliveira Neto

Uso de um Modelo de Interceptadores para Prover Adaptação Dinâmica no InteGrade

Goiânia
2008

Jesus José de Oliveira Neto

Uso de um Modelo de Interceptadores para Prover Adaptação Dinâmica no InteGrade

Dissertação apresentada ao Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação.

Orientador: Prof. Dr. Fábio Moreira Costa

Goiânia
2008

Jesus José de Oliveira Neto

Uso de um Modelo de Interceptadores para Prover Adaptação Dinâmica no InteGrade

Dissertação defendida no Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás como requisito parcial para obtenção do título de Mestre em Ciência da Computação, aprovada em 25 de Abril de 2008, pela Banca Examinadora constituída pelos professores:

Prof. Dr. Fábio Moreira Costa
Instituto de Informática – UFG
Presidente da Banca

Prof. Dr. Renato Fontoura de Gusmão Cerqueira
Departamento de Informática – PUC-Rio

Prof. Dr. André Barros de Sales
Departamento de Computação – UCG

Jesus José de Oliveira Neto

Graduou-se em Ciência da Computação pela Universidade Federal de Goiás (2004). Durante sua graduação, foi monitor da disciplina de Linguagens e Técnicas de Programação em 2003 e no ano seguinte, na disciplina de Projeto e Análise de Algoritmos. Entre 2005 e 2006, foi bolsista DTI do CNPq no Instituto de Informática da UFG. Atualmente está cursando o mestrado em Ciência da Computação pela Universidade Federal de Goiás, concentrando sua pesquisa no desenvolvimento de mecanismos adaptativos para computação em grade.

Obrigado à minha mãe, minha irmã e aos meus avós, por todo o apoio que me deram durante a elaboração desta dissertação.

Agradecimentos

À minha mãe, minha irmã e meus avós pela força que me deram desde o início.

Aos colegas do GEApIS - Grupo de Estudos Aplicados à Internet e Sistemas Distribuídos pelos momentos de descontração e amizade. Agradeço também por toda informação que obtive do laboratório do GEApIS que foi de grande importância para este trabalho.

A todos que, direta ou indiretamente, contribuíram para a realização deste trabalho.

Aquele que conhece bem os outros é um sábio, mas aquele que conhece bem a si mesmo é um iluminado.

Lao-Tsé..

Resumo

José de Oliveira Neto, Jesus. **Uso de um Modelo de Interceptadores para Prover Adaptação Dinâmica no InteGrade**. Goiânia, 2008. **??**p. Dissertação de Mestrado. Instituto de Informática, Universidade Federal de Goiás.

Grades computacionais são conjuntos de recursos computacionais que fornecem diversos tipos de serviços, tais como armazenamento e processamento, para aplicações que podem estar espalhadas por diferentes domínios administrativos. Desta forma, várias empresas e instituições acadêmicas têm interesse no seu uso para a execução de aplicações que exijam um alto poder computacional. Entretanto, grades computacionais são ambientes de execução bastante diversificados e complexos, pois possuem alta variação na disponibilidade de recursos, instabilidade de seus nós e variações na distribuição de carga, entre outros problemas. Este trabalho apresenta um modelo de interceptadores dinâmicos e seu uso no InteGrade, um *middleware* de grade oportunista. O uso de interceptadores tem por finalidade prover suporte para adaptação dinâmica no InteGrade através de seu *middleware* de comunicação, assim, contribuindo para que o mesmo tenha condições de lidar com o ambiente de execução altamente variável das grades computacionais sem que sejam necessárias alterações em sua implementação. Desta forma, este trabalho busca fornecer recursos de adaptação dinâmica para o InteGrade e não para aplicações de grade. No entanto, estas aplicações poderão se beneficiar dos recursos de adaptação oferecidos pelo InteGrade.

Palavras–chave

Interceptadores dinâmicos, middleware de grade, adaptação dinâmica, reflexão computacional.

Abstract

José de Oliveira Neto, Jesus. **Use of an Interceptor Model to provide Dynamic Adaptation in InteGrade: C**. Goiânia, 2008. **??**p. MSc. Dissertation. Instituto de Informática, Universidade Federal de Goiás.

omputer grids are sets of computational resources that provide diverse types of services, such as storage and processing, on behalf of applications spread across different administrative domains. Many companies and academic institutions have demonstrated interest in their use for the execution of applications that demand huge amounts of computation power and storage. However, computer grids are complex and diversified execution environments, which exhibit high variation of resource availability, node instability and variations on load distribution, among other problems. This work presents a model of dynamic interceptors and its use in InteGrade, an opportunistic grid middleware. The use of interceptors aims to provide dynamic adaptation in InteGrade through its communication middleware, thus contributing to make InteGrade able to deal with the highly variable execution environment of computer grids without requiring changes to its implementation. Therefore, this work aims to offer dynamic adaptation capabilities to InteGrade and not to grid applications. Nevertheless, these applications will be able to benefit from adaptation provided by InteGrade.

Keywords

Dynamic interceptors, grid middleware, dynamic adaptation, computational reflection.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos de Programas

CAPÍTULO 1

Introdução

A grande difusão das tecnologias de *middleware* [13], devido ao seu uso em diversas aplicações e serviços, resultou num considerável amadurecimento desta área e, também, na especialização das aplicações distribuídas existentes. Em particular, aplicações que precisam se adaptar ao contexto e condições do ambiente de execução têm se tornado cada vez mais comuns na computação em geral. Uma plataforma de *middleware* que possua um configuração fixa não será capaz de lidar, por exemplo, com aplicações móveis ou de multimídia distribuída que exijam um suporte com maior dinamismo, ou seja, com condições de se adequar às alterações no seu ambiente de execução e às mudanças nos seus requisitos de forma eficiente.

O dinamismo é um fator crescente em ambientes computacionais distribuídos. Cada vez mais, diferentes plataformas de hardware, tais como computadores pessoais, *laptops* e PDA's que utilizam diferentes tipos de software estarão conectadas através de redes heterogêneas formadas por segmentos Ethernet e redes sem fio de longo, médio e curto alcance. Desta forma, o dinamismo será um fator crucial para a infra-estrutura computacional do futuro.

Grades computacionais [9] são um bom exemplo de ambientes computacionais distribuídos com alto grau de dinamismo. Essas grades são constituídas por uma infra-estrutura de hardware e software capaz de interligar e compartilhar diversos recursos computacionais que podem estar distribuídos em grandes áreas geográficas. Os recursos que podem ser fornecidos pelas grades geralmente são: processamento, armazenamento de grande capacidade e até componentes de software, como bancos de dados e aplicações.

Esta infra-estrutura computacional tem chamado a atenção de grandes corporações, indústrias e instituições de pesquisa que possuem um grande parque computacional, oferecendo uma forma bastante atrativa de racionalizar e utilizar todo este poder computacional que, na maior parte do tempo, é pouco aproveitado. A grande vantagem na utilização das grades computacionais é a possibilidade de utilizar redes de computadores já existentes e de baixo custo para realização de tarefas que exijam um alto poder computacional, ao invés de adquirir uma sofisticada máquina paralela.

Entretanto, as grades computacionais são ambientes distribuídos bastante diversi-

ficados e complexos. Aspectos que evidenciam estas características são mostradas abaixo [9]:

- Heterogeneidade: Os componentes que formam a infra-estrutura tendem a ser extremamente heterogêneos. Grades computacionais possuem recursos de vários tipos, softwares de várias versões, instrumentos e serviços dos mais variados tipos.
- Alta dispersão geográfica: Grades podem estar distribuídas em escala global, agregando serviços localizados em várias partes do planeta.
- Compartilhamento: Ao contrário de *clusters* [6], uma grade pode não ter seus recursos dedicados a uma aplicação de forma exclusiva por um determinado período de tempo. Isso tem impacto no desenvolvimento de aplicações que executam sobre a infra-estrutura de uma grade computacional.
- Múltiplos domínios administrativos: Grades congregam recursos de várias instituições. Sendo assim, além da heterogeneidade mencionada anteriormente, é possível também a existência de várias políticas de acesso e uso dos serviços, de acordo com as diretrizes de cada domínio que faz parte da grade.
- Controle distribuído: Tipicamente não há uma entidade única que tenha poder sobre toda a grade. Isso é um reflexo da dispersão dos componentes da grade, pois cada instituição pode implementar sua própria política em seus recursos locais, sem interferir diretamente na implementação de políticas e no acesso aos serviços de outras instituições participantes.

As plataformas de *middleware* convencionais são capazes de oferecer às aplicações distribuídas, transparência e independência, por exemplo, dos detalhes dos protocolos de rede, linguagens de programação, sistemas operacionais e hardware. Além de lidar com a heterogeneidade presente em ambientes distribuídos, estas plataformas de *middleware* também auxiliam no tratamento de outros problemas presentes na comunicação entre aplicações distribuídas tais como sincronismo, concorrência e falhas de sistema.

Toda esta infra-estrutura de suporte utilizada para ocultar estas diferenças é então aproveitada pelos sistemas de *middlewares* de grade, que têm por objetivo a formação das grades computacionais a partir da integração de recursos ociosos disponíveis em parques computacionais já existentes.

No entanto, devido à natureza dinâmica e à alta escalabilidade das grades computacionais, o seu gerenciamento e configuração através de *middleware* de grade é bastante difícil. Para lidar com este e outros ambientes distribuídos, as plataformas de *middleware* existentes devem oferecer suporte a adaptação dinâmica.

Uma plataforma de *middleware* dinâmica pode otimizar sua configuração dependendo das variações ocorridas no ambiente de execução e nos requisitos da aplicação, ou

seja, tem capacidade de realizar inspeções e mudanças em si mesma durante a sua execução. No entanto, um modelo ideal de *middleware* dinâmico também deve ser capaz de oferecer transparência para as aplicações que não estão interessadas em detalhes internos da plataforma, ao mesmo tempo em que fornece controle fino para as aplicações que podem se beneficiar em conhecer o seu próprio ambiente de execução [31].

Interceptadores [22, 39] são uma forma de adaptação dinâmica que permite estender a funcionalidade básica fornecida por um *middleware* sem afetar a sua arquitetura. A idéia básica por trás de interceptadores é permitir a introdução e configuração de algum tipo ou qualidade de serviço ou funcionalidade extra no *middleware* em tempo de execução. Por exemplo, durante a invocação de método remoto, geralmente ocorre o envio de uma requisição de um cliente para um servidor, aceitando a requisição no servidor, gerando uma resposta do servidor para o cliente, e enviando a resposta para o cliente. Esta requisição ou resposta é então capturada por interceptadores que podem estar em diferentes pontos dentro do *middleware*, tanto no lado do cliente quanto no lado do servidor. A partir destes pontos, os interceptadores podem interferir na maneira como a requisição ou resposta é tratada para adicionar propriedades não-funcionais tais como segurança, criptografia e autenticação.

Esta dissertação apresenta um modelo de interceptadores dinâmicos que permite a uma plataforma de *middleware* convencional a capacidade de inspecionar a si e o seu ambiente de execução e, dependendo das mudanças que podem ocorrer, modificar dinamicamente seu comportamento e suas funcionalidades. Este modelo de interceptadores possui uma arquitetura composta por três componentes. O primeiro componente é o *Monitor*, responsável por verificar mudanças no ambiente de execução e no estado interno do *middleware*. O segundo componente é *Implementor*, que possui a definição de uma propriedade não-funcional que é utilizada caso o *Monitor* detecte alguma mudança. O terceiro componente é o *Context*, que fornece ao *Monitor* informações sobre o estado interno do *middleware*.

Este modelo de interceptadores foi aplicado no InteGrade [7], um *middleware* de grade oportunista, que tem por objetivo aproveitar os recursos ociosos de computadores pessoais para sua utilização em tarefas de computação de alto desempenho. Caso seja empregado em uma rede de computadores existente, não necessita de novos equipamentos de hardware para ser executado. Dessa forma, instituições como universidades ou centros de pesquisas podem utilizar os recursos computacionais que já possuem para a execução de aplicações de grade. Outras vantagens do InteGrade são apresentadas abaixo:

- Por ser um *middleware* executado sobre sistemas operacionais pré-existentes, não há necessidade de alterar a instalação de software das máquinas.
- Possui suporte a aplicações paralelas que demandam comunicação entre seus nós, além de aplicações trivialmente paralelizáveis, onde cada nó da aplicação não se

comunica com os demais.

- Não degrada o desempenho das máquinas que fornecem recursos à grade.

A introdução de interceptadores no InteGrade tem o intuito de facilitar a inserção e configuração de serviços que possam ser ativados dinamicamente, oferecendo, portanto, um suporte mais apropriado de acordo com as necessidades das aplicações em tempo de execução.

1.1 Motivação

O InteGrade está ainda em fase de desenvolvimento e, por isso, apenas os componentes mais essenciais para o funcionamento deste sistema foram implementados. Por este motivo, o InteGrade ainda é incapaz de lidar com a alta variação na disponibilidade de recursos, instabilidade dos nós de uma grade, variações de sobrecarga nos nós e nas redes e outros problemas presentes em grades computacionais.

Desta forma, a adição de mecanismos de adaptação dinâmica ao InteGrade é necessária para lidar com a alta dinamicidade presente nas grades computacionais. Os ORBs (*Object Request Broker*) JacOrb [55] e OiL [38], utilizados pelo InteGrade como suporte para abstrair os detalhes de comunicação entre os seus componentes, são um bom ponto de partida para a inclusão de mecanismos de adaptação dinâmica.

Interceptadores dinâmicos são um mecanismo de adaptação dinâmica que possibilita estender a funcionalidade básica fornecida por um ORB, com alterações mínimas em sua implementação. Com o uso de interceptadores, o InteGrade vai ser capaz de alterar dinamicamente o modo como os seus componentes interagem entre si através de seus ORBs de comunicação. Além disso, o InteGrade poderá usufruir deste mecanismo de adaptação dinâmica de forma transparente sem causar impacto na sua arquitetura.

No entanto, um fato importante a ser observado nesta abordagem é que ao inserir interceptadores, por exemplo, somente no OiL, apenas os componentes do InteGrade que utilizam este ORB irão se beneficiar diretamente deste mecanismo de adaptação dinâmica. Por causa disso, para que todos os componentes do InteGrade possam usufruir do uso de interceptadores, este mecanismo de adaptação dinâmica deve ser inserido também no JacORB.

1.2 Objetivos do Trabalho

Este trabalho tem por finalidade investigar a factibilidade e os benefícios do uso de um modelo de interceptadores para prover ao InteGrade a capacidade de se adaptar

dinamicamente através dos seus ORBs de comunicação. Os principais objetivos deste trabalho são apresentados abaixo:

- Implementação de um modelo geral de interceptadores dinâmicos e seu uso para permitir adaptação dinâmica de aspectos comportamentais dos serviços e infra-estrutura do InteGrade.
- Identificar quais aspectos do comportamento e funcionalidades do InteGrade podem se beneficiar da adaptação dinâmica ao utilizar interceptadores dinâmicos.
- Desenvolvimento de experimentos práticos de forma a verificar a capacidade do InteGrade em lidar com o dinamismo presente nas grades computacionais através do uso de interceptadores dinâmicos.

O restante desta dissertação está organizado da seguinte forma. No Capítulo 2 são abordados alguns fundamentos importantes para o entendimento deste trabalho, descrevendo plataformas de *middleware* convencionais, plataformas de *middleware* de grades e reflexão computacional, com seus principais conceitos e exemplos de implementação. O Capítulo 3 apresenta o modelo de interceptadores dinâmicos, mostrando detalhes de sua arquitetura e também da sua implementação. O Capítulo 4 demonstra algumas aplicações do modelo de interceptadores que foram realizados durante o processo de desenvolvimento a fim de definir os benefícios trazidos pelo seu uso para prover adaptação dinâmica ao InteGrade. Finalmente, o Capítulo 5 traz as conclusões sobre o trabalho desenvolvido, os resultados obtidos e os possíveis trabalhos futuros.

CAPÍTULO 2

Middleware de Grade e Adaptação Dinâmica

Neste capítulo são apresentados os fundamentos mais importantes para o entendimento deste trabalho. Inicialmente, descrevemos o conceito de plataformas de *middleware* convencionais, utilizadas por algumas plataformas de *middleware* de grade, inclusive o InteGrade, como um mecanismo para abstrair os detalhes de comunicação entre seus componentes, tratando de problemas presentes nos ambientes distribuídos das grades computacionais tais como heterogeneidade, acesso e falhas de sistema.

Em seguida, são apresentados os principais aspectos de um *middleware* de grade e também alguns trabalhos existentes nesta área. Logo depois, apresentamos as principais características de reflexão computacional que têm sido adotadas por diversos tipos de plataformas de *middleware*, incluindo as plataformas de *middleware* de grade, como forma de oferecer recursos de adaptação dinâmica. Dando seqüência, descrevemos algumas plataformas de *middleware* que utilizam mecanismos de reflexão computacional para se adaptarem às variações que ocorrem em ambientes distribuídos altamente dinâmicos, sendo conhecidas como plataformas de *middleware* reflexivo. Por último, descrevemos alguns trabalhos que demonstram o uso de mecanismos reflexivos por plataformas de *middleware* de grade.

2.1 Plataformas de *Middleware* Convencionais

As primeiras tecnologias de *middleware* surgiram da necessidade de se criar uma infra-estrutura para facilitar o desenvolvimento de aplicações distribuídas. Devido ao avanço das tecnologias de redes, foi possível que uma aplicação tivesse seus componentes distribuídos ao longo de uma rede de computadores e comunicando entre si com um nível de qualidade suficiente para se apresentar como um sistema único e coerente. O uso desta abordagem permitiu a criação de aplicações a um baixo custo e com um maior desempenho. Os maiores benefícios oferecidos pelas aplicações distribuídas são o compartilhamento de recursos, maior disponibilidade de serviços e melhor escalabilidade.

Entretanto, as aplicações distribuídas possuem complicações no seu desenvolvimento que não ocorrem nas aplicações centralizadas. Dentre os maiores problemas, podemos citar:

- *Heterogeneidade*: os computadores e outros dispositivos que compõem uma rede podem ser de vários tipos de hardware e utilizar diferentes sistemas operacionais.
- *Acesso*: os protocolo de comunicação utilizados para acessar um computador ou dispositivo podem ser diferentes de uma rede para outra. Além disso, os mecanismos internos, a sintaxe e a semântica das linguagens de programação utilizadas pelos programas ou serviços que constituem essas redes também podem variar.
- *Falhas parciais de sistema*: dentro de uma rede, um computador ou dispositivo pode parar de funcionar repentinamente por algum motivo, que pode ser queda de energia, defeito, entre outros.

Para lidar com estas dificuldades, uma plataforma de *middleware* precisa abstrair os detalhes de comunicação entre os componentes que compõem uma aplicação distribuída. Como mostra a Figura 2.1, as plataformas de *middleware* geralmente são posicionadas entre as aplicações e os sistemas operacionais subjacentes. Assim, o *middleware* pode oferecer uma *interface* de comunicação comum para os componentes da aplicação que possibilite ocultar uma falha de sistema ou os detalhes quanto aos diferentes sistemas operacionais e protocolos de rede usados num ambiente distribuído e heterogêneo.

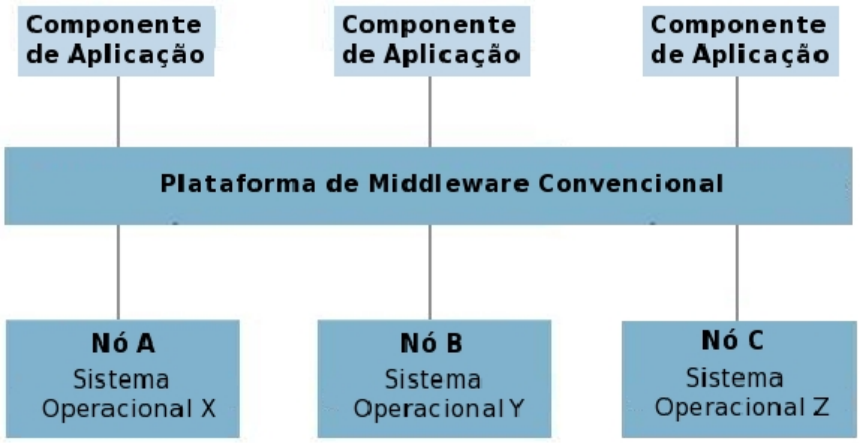

Figura 2.1: *Uma plataforma de middleware convencional*

Através do uso de plataformas de *middleware*, o nível de complexidade no desenvolvimento de uma aplicação distribuída é bastante reduzido. Desta forma o desenvolvedor pode se concentrar mais nos problemas específicos da aplicação, já que a maior parte dos problemas de distribuição serão tratados transparentemente pela plataforma [12].

2.2 Plataformas de *Middleware* de Grade

Plataformas de *middleware* de grade [21, 20, 4] são sistemas que buscam disponibilizar serviços computacionais sob demanda de forma transparente através da integração e do gerenciamento de recursos ociosos disponíveis em parques computacionais já instalados. Geralmente, o recurso mais utilizado é o processamento. Porém, outros recursos, tais como armazenamento em disco ou aplicações também são requisitados. Os principais aspectos que são comuns aos vários sistemas de *middleware* de grade são listados abaixo [24]:

- *Podem integrar recursos distribuídos e descentralizados*: boa parte das plataformas de *middleware* de grade são capazes de integrar recursos dispersos em múltiplos domínios administrativos compostos por diversas tecnologias de hardware e software. Para isso, um *middleware* de grade inclui mecanismos que lidam com esta diversidade. Alguns, como o InteGrade [7], utilizam plataformas de *middleware* convencionais para abstrair as diferenças presentes nestes domínios.
- *Não substituem sistemas operacionais*: diferente dos sistemas operacionais distribuídos, um *middleware* de grade não substitui os sistemas operacionais das máquinas que compõem uma grade computacional. Na verdade, um *middleware* de grade utiliza os serviços do sistema operacional para que possa prover os seus próprios serviços e aplicações aos usuários da grade.
- *Adaptabilidade às políticas locais*: mesmo integrando recursos dispersos por vários domínios administrativos, um *middleware* de grade deve se adaptar às políticas e restricões de uso de cada um desses domínios. Desta forma, ele deve estar preparado, por exemplo, para situações nas quais os recursos compartilhados por um domínio são realocados para atender a algum usuário local.

Os sistemas de *middleware* de grade podem ser classificados de acordo com a atividade principal à qual se destinam. Algumas dessas categorias são:

- Grade Computacional (*Computing Grid*): *middleware* de grade cujo objetivo principal é a integração de recursos computacionais dispersos para prover diversos serviços de forma transparente aos seus usuários. Uma subcategoria das grades computacionais são as grades computacionais oportunistas (*Scavenging Grids ou High Throughput Computing Grids* [35]), que fazem uso da capacidade computacional ociosa de recursos não dedicados à grade. O InteGrade e o Condor são exemplos desta subcategoria.
- Grade de Dados (*Data Grid*): *middleware* de grade cujo objetivo principal é permitir o acesso, pesquisa e classificação de grandes volumes de dados espalhados em diversos banco de dados.

- Grade de Serviços (*Service Grid*): fornece serviços viabilizados pela integração de diversos recursos computacionais, como por exemplo um ambiente para colaboração à distância.

Dentre os projetos de *middleware* de grade existentes, podemos destacar: o InteGrade, que se encontra dentro do escopo deste trabalho, o Globus [23, 18], por ser o mais conhecido e por ter influenciado inúmeros outros trabalhos e, por fim, o Condor [10] pela característica oportunista na exploração de recursos. Estes sistemas de *middleware* de grade são do tipo Grade Computacional e são descritos a seguir.

2.2.1 InteGrade

O InteGrade [7] é um *middleware* de grade oportunista que fornece suporte para o desenvolvimento de aplicações que utilizam recursos ociosos disponíveis em parques computacionais já instalados. Geralmente boa parte dos computadores pessoais permanecem totalmente ociosos durante longos intervalos de tempo: por exemplo, os laboratórios de ensino reservados aos alunos de um instituição acadêmica são pouco utilizadas durante a noite. Portanto, a criação de uma infra-estrutura de software que utilize de forma efetiva esses recursos, que de outra forma seriam desperdiçados, permitiria uma maior economia para as instituições que demandam grande poder computacional. As principais características do InteGrade são mostradas abaixo:

- aproveitamento o poder computacional ocioso das máquinas em um parque computacional já instalado.
- arquitetura orientada a objetos.
- provisão de suporte para aplicações paralelas.
- análise e monitoramento dos padrões de uso das máquinas para melhorar o desempenho do escalonador.
- não degradação do desempenho das máquinas que fornecem recursos à grade.

O InteGrade utiliza os ORBs JacOrb e OiL, que são duas plataformas de *middleware*, como suporte para abstrair os detalhes de comunicação entre os seus componentes. Um dos principais motivos da escolha destes ORBs foi o fato de ambos terem sido implementados a partir do padrão CORBA. Os ORBs baseados em CORBA permitem a integração de módulos escritos nas mais diferentes linguagens de programação, executando sobre diversas plataformas de hardware e software. Além disso, fornecem uma série de serviços, tais como o *Serviços de Nomes* [47] e Negociação (*Trading*) [45], os quais são utilizados pelo InteGrade. No entanto, estes serviços são básicos já que não oferecem recursos importantes, tais como segurança.

Atualmente, a arquitetura do InteGrade está definida através de uma estrutura hierárquica onde as unidades estruturais são os aglomerados (*clusters*), que são conjuntos de máquinas agrupadas de acordo com um certo critério como, por exemplo, um mesmo domínio administrativo.

Cada uma das máquinas pertencentes ao aglomerado é chamada de nó, existindo vários tipos de nós, conforme o papel desempenhado. O InteGrade é dividido em apenas dois tipos de nó: os nós *Gerenciadores de Recursos do Aglomerado*, que armazenam os módulos que são responsáveis por coletar informações, tais como escalonamento ou disponibilidadede recursos de módulos que estão em outros nós, sendo também responsáveis pela comunicação com gerenciadores de outros aglomerados; e os nós *Provedores de Recursos*, que são, em geral, máquinas ou conjuntos de máquinas (*cluster*) que fornecem seus recursos ociosos (ou então todos os recursos disponíveis caso sejam nós dedicados) e pelas quais se submete as aplicações para serem executadas na grade. Esta arquitetura [7] e seus módulos são ilustrados na Figura 2.2 e descritos abaixo:

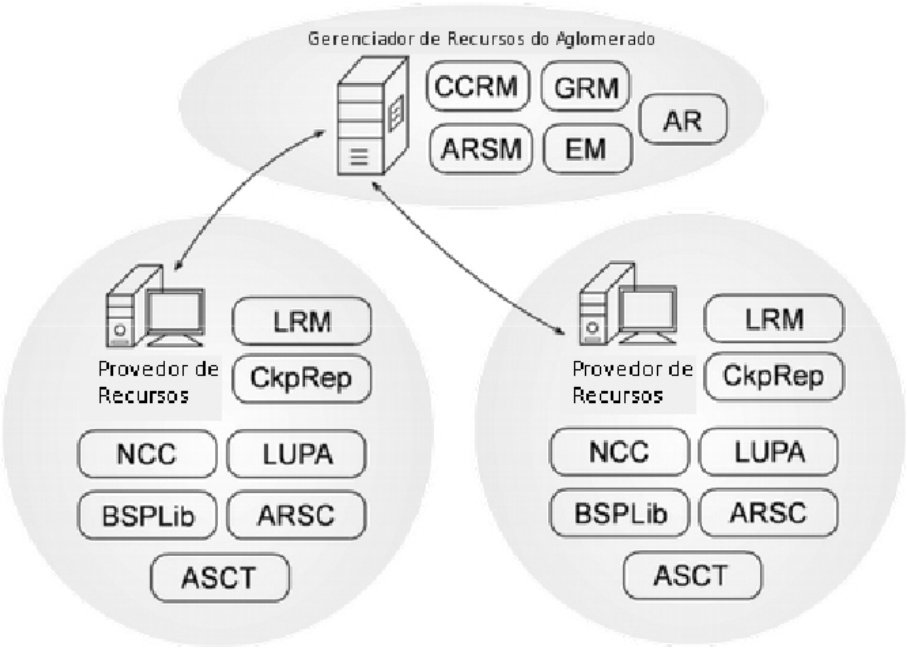

Figura 2.2: *Arquitetura do InteGrade [7]*

- O *AR* (*Application Repository*) armazena as aplicações que os usuários submetem para serem executadas na grade. O registro da aplicação no repositório é feito através do *ASCT*. Para o *LRM* poder executar tal aplicação, ele deve fazer a sua requisição no repositório. O repositório de aplicações também fornece outras funções mais avançadas, como por exemplo o registro de múltiplos binários para a mesma aplicação, para que uma mesma possa executar em diferentes plataformas. Outras duas opções disponíveis são a categorização de aplicações, otimizando a busca de aplicações no repositório e o controle de versões.

- O *ARSM* (*Application Repository Security Manager*) gerencia a segurança em um *cluster* da grade, fornecendo suporte para assinaturas digitais, encriptação de dados, autenticação e autorização.
- O *ARSC* (*Application Repository Security Client*): auxilia os componentes locais do InteGrade a interagirem com os componentes remotos de uma maneira segura.
- O *EM* (*Execution Manager*) mantém informações sobre a execução de uma aplicação, tais como o início e o término da execução, o motivo de seu término, que pode ser devido a alguma falha interna, bem como os nós da grade nas quais a aplicação está executando. Este módulo também coordena a reinicialização e migração das aplicações.
- O *GRM* (*Global Resource Manager*), recebe dos *LRMs* atualizações sobre a disponibilidade de recursos em cada nó do aglomerado. Desta forma, é possível saber se uma aplicação submetida pelo usuário pode ser executada ou não naquele momento. O *GRM* é responsável também pelo escalonamento de requisições de execução de aplicações no InteGrade.
- O *LRM* (*Local Resource Manager*) é executado nos nós compartilhados e dedicados do InteGrade para poder coletar e distribuir as informações referentes à disponibilidade de recursos destes nós em um determinado momento.
- O *ASCT* (*Application Submission and Control Tool*) é o módulo através do qual o usuário submete as aplicações que vão ser executadas pela grade. O usuário tem a opção de determinar quais os requisitos para executar a aplicação, como por exemplo a quantidade mínima de memória ou a plataforma de hardware e/ou software. Existe também a opção do usuário monitorar e controlar o andamento da execução da aplicação. Ao término da execução, o usuário pode recuperar os arquivos de saída de suas aplicações, caso haja algum, além das saídas de erro. Desta forma, o usuário pode determinar quais os problemas da aplicação.
- O *BSPLib* (*BSP Library*): Biblioteca que implementa a API da *BSPLib*, definida em [57], permitindo a execução de aplicações *BSP* no InteGrade.
- O *CkpRep* (*Checkpoint Repository*): armazena dados sobre o *checkpointing* de uma aplicação, que pode ser simples ou paralela do tipo BSP ou MPI, em um conjunto de nós na grade. Cada provedor de recursos que executa um *LRM* é um nó em potencial para hospedar uma instância deste módulo.
- O *CDRM* (*Cluster Data Repository Manager*) gerencia os *CkpReps* do seu *cluster*.
- O *LUPA* (*Local Usage Pattern Analyzer*) é responsável por determinar quais os períodos de tempo em que um nó possui maior disponibilidade de recursos. Esta análise é feita a partir das informações periodicamente coletadas pelo *LRM*. O *LUPA*

então armazena longas séries de dados e aplica algoritmos de *clustering* [28, 2, 49], de modo a definir as categorias comportamentais deste nó e determinar quais são seus padrões de uso.

- O *NCC* (*Node Control Center*) é executado nos nós compartilhados e permite que o usuário controle o quanto dos recursos de sua máquina pode ser compartilhado com o InteGrade.

Os diversos módulos do aglomerado InteGrade colaboram de maneira a desempenhar funções importantes no sistema. Em seguida, descrevemos os dois principais protocolos intra-aglomerado: o *Protocolo de Disseminação de Informações* e o *Protocolo de Execução de Aplicações*.

O *Protocolo de Disseminação de Informações* tem por objetivo manter o *GRM* informado sobre os recursos disponíveis nas máquinas de um aglomerado. Existem dois tipos de informação que são transmitidos para o *GRM*: estática e dinâmica. As informações estáticas são a arquitetura da máquina, versão do sistema operacional e tamanho do disco rígido e memória. Já as informações dinâmicas são a porcentagem de CPU ociosa, quantidades disponíveis de espaço no disco e na memória, entre outros. Quando um usuário submete um aplicação para ser executada, uma lista de requisitos deve ser passada para o *GRM*. Então o *GRM* utiliza as informações obtidas a partir do protocolo de disseminação de informações para procurar por uma ou mais máquinas adequadas para executar esta aplicação.

Para manter o *GRM* atualizado, os *LRMs* devem enviar informações sobre a disponibilidade de recursos a cada intervalo de tempo t1. Esta atualização também serve para que o GRM detecte a queda de algum *LRM*. Entretanto, se houver alguma mudança significativa (por exemplo, 15 % de variação no uso de memória) num intervalo de tempo t2, onde t2 < t1, o *LRM* também deve relatar ao *GRM* sobre esta alteração.

O *Protocolo de Execução de Aplicações* do InteGrade tem por objetivo fornecer meios para que um usuário da grade submeta aplicações para execução sobre recursos compartilhados. Os passos deste protocolo são ilustrados na Figura 2.3 e descritos abaixo:

1. Através do *ASCT*, o usuário solicita a execução de uma aplicação que tenha sido registrada no repositório de aplicações. Caso seja necessário, o usuário pode especificar requisitos para a execução de sua aplicação, como por exemplo a velocidade mínima do processador ou a versão do sistema operacional.

2. Assim que o *GRM* recebe a requisição de execução, começa então a procura por um nó candidato ou então mais de um nó, caso seja uma aplicação paralela, para executar a aplicação com base nas informações fornecidas pelo protocolo de disseminação de informações. Se não houver um nó disponível, o *GRM* notifica tal fato ao *ASCT*.

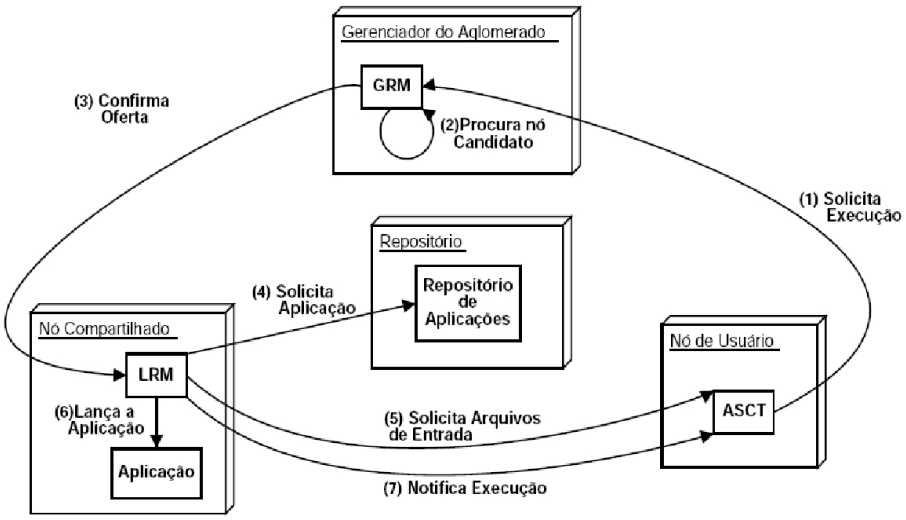

Figura 2.3: *Protocolo de Execução de Aplicações do InteGrade [24]*

3. Ao encontrar um nó ou um cojunto de nós, no caso de ser uma aplicação paralela, que satisfaça os requisitos da aplicação, o *GRM* envia ao *LRM* da máquina candidata a solicitação para executar a aplicação.

4. Após confirmar, de fato, que possui os requisitos disponíveis para executar a aplicação, o *LRM* solicita a aplicação para o repositório de aplicações. Esta confirmação fornecida pelo LRM é necessária pois pode haver alterações na disponibilidade de recursos no intervalo de tempo entre a busca por nós candidatos e a solicitação do *GRM* para executar a aplicação. Se houver realmente uma mudança nos recursos disponíveis, o *LRM* deve notificar tal fato ao *GRM*, que retorna ao passo 2 e recomeça a busca por um nó candidato.

5. Após receber a aplicação do repositório, o *LRM* solicita os eventuais arquivos de entrada ao *ASCT* requisitante.

6. Com todos os dados necessários, o *LRM* então lança a aplicação.

7. O *ASCT* recebe uma notificação informando que sua requisição foi atendida. Através desta notificação, o *ASCT* sabe em qual *LRM* a aplicação está sendo executada ou então em quais *LRMs*, caso a aplicação seja paralela, podendo assim controlá-la remotamente.

O InteGrade está ainda em fase de desenvolvimento e, por isso, existem diversos trabalhos sendo feitos para adicionar novas funcionalidades visando tornar o sistema mais eficiente e corrigir as falhas atuais. O trabalho apresentado nesta dissertação constitui um destes trabalhos e tem por objetivo permitir que o InteGrade tenha suporte a adaptação dinâmica através de um modelo de interceptadores dinâmicos.

2.2.2 Globus

O Globus [23, 18] é atualmente um dos projetos de maior importância na área de computação em grade. O seu desenvolvimento foi baseado no I-WAY [17], uma infra-estrutura de Grade que era composta por 11 redes de alta velocidade e 17 institutos de pesquisa. Essa infra-estrutura foi criada para funcionar temporiaramente durante o congresso *Supercomputing '95*. Devido ao sucesso obtido naquele congresso, surgiu a iniciativa da criar o projeto Globus.

O *Globus Toolkit* [23, 19, 21], desenvolvido dentro do projeto Globus, é um conjunto de ferramentas e bibliotecas de software que dão suporte ao desenvolvimento de aplicações para sistemas de computação em grade, assim como para a implantação de tais sistemas. O projeto é mantido por uma comunidade de programadores e é baseado no uso de padrões e componentes de software de código aberto. Com o uso desta ferramenta, é possível implementar recursos tais como segurança, busca de informações, gerenciamento de recursos e de dados, comunicação, detecção de falhas e alocação de recursos.

A primeira versão do *Globus Toolkit* foi lançada em 1999. No entanto, foi apenas em 2001 que o mercado percebeu todas as possibilidades que esta ferramenta poderia oferecer. Depois da segunda versão, a arquitetura do *Globus Toolkit* se tornou um padrão para a computação em grade.

A Figura 2.4 apresenta os componentes da segunda versão do *Globus Toolkit*. Estes componentes podem ser usados, tanto independentemente quanto em conjunto, no desenvolvimento de aplicações e de ferramentas para grades. Os principais serviços presentes na infra-estrutura são: *Globus Resource Allocation Manager (GRAM)*, *Monitoring and Discovery Service (MDS)*, *Reliable File Transfer (RFT)* e *Grid Security Infrastructure (GSI)*.

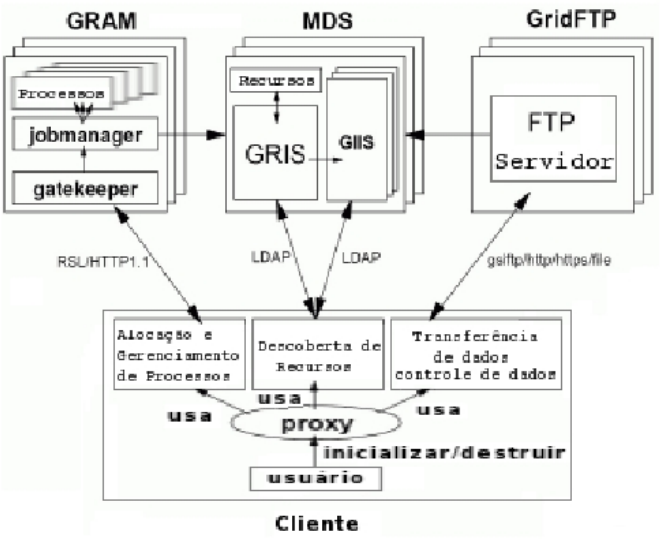

Figura 2.4: *Componentes da segunda versão do Globus Toolkit [23]*

O *GRAM* é o serviço responsável por alocar recursos e gerenciar as tarefas, agindo como uma ponte entre o sistema Globus e o gerenciador de recursos local. O *GRAM* é encarregado de administrar um conjunto de máquinas, sejam elas máquinas paralelas, aglomerados ou estações de trabalho. As principais funções do *GRAM* são as seguintes:

- Receber requisições solicitando recursos. Estas requisições podem ser aceitas ou negadas com base na disponibilidade de recursos e nas credenciais dos usuários que as enviaram. Estas credencias são um tratamento de segurança fornecido pelo componente *GSI*, que será descrito mais adiante.
- Fornecer uma *interface* que permita ao usuário monitorar e gerenciar os processos criados a partir de suas requisições.
- Manter o *MDS* atualizado com relação à disponibilidade dos recursos na grade.

O *MDS* é o serviço responsável por disponibilizar informações sobre os recursos disponíveis na grade. O *MDS* armazena informações sobre os vários recursos da grade, tais como o sistema operacional, a memória disponível, o espaço em disco, entre outros. Ele é constituído por dois serviços internos: *GRIS* e *GIIS*.

O *GRIS* gerencia as informações sobre um determinado recurso. Se este recurso for um computador, o *GRIS* fornece informações, por exemplo, sobre a arquitetura de hardware e sistema operacional. Caso o recurso seja uma rede, o *GRIS* deve informar sobre o seu tipo, a velocidade máxima de transmissão e a largura de banda disponível no momento. Uma instância do *GRIS* pode representar uma única máquina da grade ou então um aglomerado inteiro composto por diversos computadores. Já o *GIIS* reúne as informações dos vários *GRIS* espalhados na grade em único servidor.

O *RFT* é o serviço responsável pela transferência de arquivos entre as máquinas da grade de forma segura.

O *GSI* [25] é o serviço responsável por prover segurança dentro do Globus. Os mecanismos para autenticação e comunicação segura fornecidos por este serviço são baseados em certificados X.509, infra-estrutura de chave pública (PKI), protocolos SSL/TLS e certificados *proxy* X.509. Os recursos de segurança fornecidos pelo GSI são descritos abaixo:

- Autenticação única: o usuário precisa se autenticar apenas uma vez no sistema para ter acesso aos recursos da grade.
- Comunicação segura: impede que uma entidade desconhecida tenha acesso às comunicações das aplicações de um usuário credenciado.
- Delegação: permite delegar parte das permissões de um usuário a uma outra entidade, por exemplo, uma aplicação. Assim, esta aplicação pode requisitar mais recursos sem necessidade de uma nova autenticação.

Na terceira versão do *Globus Toolkit*, ilustrada na Figura 2.5, houve mudanças profundas com relação às versões anteriores. Estas modificações se devem ao fato de que o *Globus Toolkit* passou a ser baseado em uma arquitetura e um conjunto de padrões definidos por um comitê chamado *Open Grid Forum* [42]. A utilização de padrões é importante pois permite que o *middleware* de grade possa ser utilizado em larga escala. A arquitetura e os padrões definidos por este comitê são mostrados a seguir.

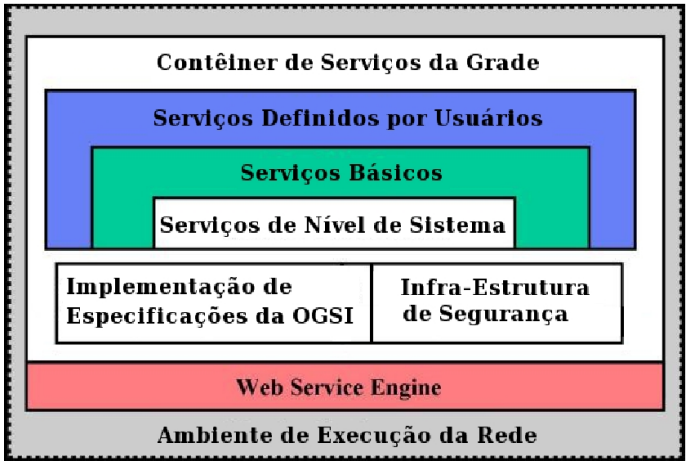

Figura 2.5: *Globus Toolkit na versão 3 [23]*

Web Services [54]: é o padrão utilizado no desenvolvimeno de *Grid Services* e, desta forma, a base para a OGSA [43], para a *OGSI* [44] e também para a terceira versão do *Globus Toolkit*. *Web Services* é uma tecnologia de computação distribuída que possibilita a comunicação entre aplicações usando protocolos e linguagem abertos e amplamente conhecidos como o HTTP e o XML.

Open Grid Services Architecture (OGSA) [43]: A OGSA tem por objetivo definir quais serviços um ambiente de grade pode oferecer. Os serviços definidos em *OGSA* são chamados de *Grid Services*, em referência aos *Web Services* nos quais foram baseados. Os *Grid Services* são expressos em uma linguagem de definição de *interface* conhecida como WSDL (*Web Services Description Language*) [8], utilizada na especificação de *Web Services* em geral. Assim como outras linguagens de definição de *interface*, como por exemplo a IDL (*Interface Definition Language*) do padrão CORBA, WSDL não especifica modelo, arquitetura ou linguagem de programação. Os *Grid Services* são compostos por um ou mais *portTypes*, uma espécie de *sub-interface* de *Web Service* que possui uma funcionalidade específica.

Open Grid Services Infrastructure (OGSI) [44]: a *OGSI* tem por objetivo definir como construir, gerenciar e expandir um *Grid Service*. Para isso, a *OGSI* provê um conjunto de *portTypes* que podem ser inseridos num *Grid Service* de modo a fornecer alguma funcionalidade básica. Os principais tipos de *portTypes* providos pela *OGSI* são listados a seguir.

- *GridService*: apresenta informações sobre o *Grid Service* em questão, permitindo gerenciar o seu ciclo de vida. Este *portType* deve ser definido obrigatoriamente por todos os *Grid Services*.
- *Factory*: utilizado para criar instâncias de um *Grid Service*.
- *NotificationSource*: permite que clientes se registrem junto a um *Grid Service* para receberem notificações, tornando assim o *Grid Service* orientado a eventos *(publish/subscribe)*.
- *NotificationSubscription*: permite o gerenciamento de propriedades referentes ao registro junto a um *Grid Service*.

Na quarta versão do *Globus Toolkit* [26], a especificação WSRF (*Web Service Resource Framework*) [14], derivada da especificação *OGSI*, passou a ser utilizada devido à necessidade de interoperabilidade com *web services* em geral. Através destas mudanças, o *OGSI* pôde ser incluído como parte dos padrões de *serviços web* e o *OGSA* então passa a ser baseado diretamente nestes padrões, conforme ilustra a Figura 2.6.

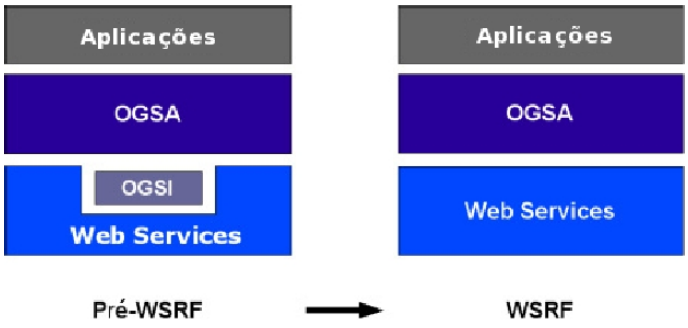

Figura 2.6: *OGSI para WSRF [26]*

O projeto Globus, devido à sua estrutura e padrões utilizados, têm servido de referência para o estudo de grades computacionais e também para a criação de outros projetos de *middleware* de grade. Isto o torna um dos projetos de maior impacto na computação em grade.

2.2.3 Condor

Condor [10] é um *middleware* de grade desenvolvido na Universidade de Wisconsin-Madison. Seu uso é indicado para usuários que precisam de um grande poder computacional ao longo de um grande período de tempo, como dias, semanas ou meses [34]. Geralmente, as necessidades de recursos deste tipo de usuário superam em muito a capacidade disponível. Por isso, esta categoria de uso é conhecida como *High Throughput*

Computing [35]. Para atender a esta categoria, Condor disponibiliza recursos computacionais buscando manter um desempenho sustentável, mesmo que haja variações dentro da grade computacional.

Atualmente, Condor funciona tanto para grades dedicadas quanto para grades oportunistas. A estrutura do Condor está organizada na forma de aglomerados de máquinas, chamados de *Condor Pools*. Normalmente, cada aglomerado pertence a um domínio administrativo distinto, sendo independentes entre si. Entretanto, tal organização não é obrigatória. A arquitetura de um aglomerado Condor é ilustrada na Figura 2.7 e descrita abaixo.

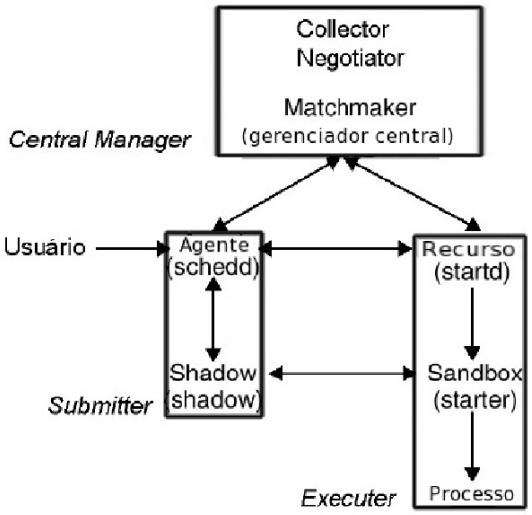

Figura 2.7: *Arquitetura de um Condor Pool [10]*

O *Submitter* representa um nó cliente da grade. Possui um módulo interno chamado *Schedd*, que permite ao usuário solicitar a execução de uma aplicação ao sistema Condor. Portanto, este módulo deve estar presente em todas as máquinas a partir das quais se deseja submeter aplicações para execução. Este componente serve também para o usuário monitorar e controlar remotamente a execução da aplicação.

Os *Executers* são os nós que disponbilizam recursos computacionais ao aglomerado. Nestes nós, deve estar presente o módulo interno *Startd*, que permite ao sistema Condor executar as aplicações dentro dessas máquinas. Além disso, o Startd também publica informações sobre o uso e a disponibilidade de recursos da máquina em que está inserido.

O *Central Manager* é responsável pelas tarefas de gerenciamento do aglomerado. É constituído por dois módulos internos: *Collector* e *Negotiator*. O *Collector* serve para armazenar as informações do aglomerado e receber requisições de execução transmitidas por algum de seus nós. Geralmente, uma instância deste módulo está presente em cada aglomerado. Para se manter atualizado quanto à disponibilidade de recursos da grade,

o *Collector* recebe periodicamente informações do módulo *Startd*. As requisições de execução solicitadas pelo usuário são passadas para o *Collector* através do módulo *Schedd*, que informa quais são as necessidades da aplicação. O *Negotiator* tem a função de escalonar as requisições no aglomerado. Periodicamente, o *Negotiator* verifica se existem requisições através de consultas ao *Collector*. Caso existam, o *Negotiator* então, baseado nas informações fornecidas pelo Collector, transfere as requisições para máquinas em condições de executá-las.

Assim como o InteGrade, o Condor fornece suporte ao modo de operação oportunista, o que possibilita um uso mais eficiente dos parques computacionais, sem prejudicar os usuários que utilizam essas máquinas. Este aspecto possibilita que o Condor seja cada vez mais utilizado como forma de prover recursos para uma grade computacional.

2.3 Adaptação Dinâmica

Atualmente, os ambientes computacionais distribuídos apresentam um grau de dinamismo cada vez maior. Grades computacionais são um bom exemplo deste tipo de ambiente de execução constituído por diferentes recursos de hardware e software. Esses recursos geralmente são fornecidos através da interconexão de máquinas paralelas, *clusters*, ou dispositivos móveis. Uma das características mais importantes das grades computacionais é o fato de que a disponibilidade dos recursos oferecidos pode variar mesmo durante a execução de uma aplicação. Além disso, os recursos alocados podem ser liberados e então realocados à medida em que aplicações são iniciadas e encerradas dentro da grade. Desta forma, a utilização destes recursos pelas aplicações não pode ser feita de forma estática e as mudanças que ocorrem durante a alocação de recursos não podem ser consideradas como falha.

Para lidar com o dinamismo presente nesses ambientes distribuídos, uma nova estruturação deve ser feita nas plataformas de *middleware* convencionais para que sejam capazes de oferecer suporte a adaptação dinâmica.

2.3.1 Reflexão Computacional

Para que uma plataforma de *middleware* seja dinâmica, várias abordagens têm sido utilizadas, sendo que a maioria emprega técnicas de reflexão computacional. A idéia básica da reflexão computacional define que um sistema, para ser reflexivo, deve manter uma representação interna da sua própria configuração [36]. Caso ocorra alguma mudança nesta representação, o sistema em si também sofrerá alterações. Da mesma forma, se o sistema for modificado, a representação de sua configuração também será alterada. Por causa deste processamento introspectivo, um sistema reflexivo oferece a possibilidade de inspeções e

mudanças em si mesmo durante a sua execução [31]. As principais características de um sistema reflexivo são apresentadas abaixo [12]:

- *Reificação*: a capacidade de mostrar a representação interna de um sistema através de entidades programáveis que podem ser manipuladas em tempo de execução. Absorção é o processo inverso da reificação e consiste em refletir as mudanças realizadas nesta representação no sistema real. A reificação e a absorção definem a conexão causal entre o próprio sistema e sua representação.
- *Arquitetura de meta-níveis*: um sistema reflexivo possui uma arquitetura de meta-nível quando sua estrutura é composta de um nível base, que lida diretamente com as funcionalidades do sistema, e um meta-nível, que lida com o processamento reflexivo.
- *Meta-Objeto e Protocolo de Meta-Objetos (MOP)*: nos sistemas reflexivos orientados a objetos, as entidades presentes no meta-nível são chamadas de meta-objetos. O Protocolo de Meta-Objetos é utilizado na interação com os meta-objetos e também define o tipo de reflexão fornecido pelo sistema.
- *Reflexão Estrutural*: habilidade de um sistema de fornecer uma reificação completa da sua estrutura interna em tempo de execução. Desta forma, o desenvolvedor pode inspecionar ou mudar a funcionalidade do programa e a maneira como ele interage com seu domínio.
- *Reflexão Comportamental*: habilidade de um sistema para fornecer uma representação completa da sua própria semântica em termos dos aspectos internos do seu ambiente de execução. Com isso, o programador pode inspecionar ou mudar o modo como o ambiente processa o programa, por exemplo, com relação às propriedades não-funcionais e gerenciamento de recursos. No caso de plataformas de *middleware* dinâmicas, um mecanismo comumente utilizado para implementar reflexão comportamental refere-se ao uso de *interceptadores*, que permitem modificar o comportamento de uma plataforma de *middleware* através da adição ou remoção de propriedades não-funcionais.

2.3.2 A Linguagem Lua

A reflexão computacional foi inicialmente utilizada em linguagens de programação e, desde então, tem sido empregada com sucesso em outras áreas, inclusive em sistemas distribuídos. Lua [27] é um exemplo de linguagem de programação reflexiva. A seguir, são apresentadas algumas das características da linguagem Lua importantes para o entendimento do modelo de interceptadores dinâmicos definido neste trabalho.

O carregamento e ativação de código em tempo de execução é uma característica reflexiva da linguagem Lua que permite a um sistema alterar a sua configuração ou estrutura dinamicamente. Esta característica é bastante comum em linguagens interpretadas tais como Lisp [53], sendo que a própria linguagem Lua também é interpretada. Através deste recurso oferecido pela linguagem Lua, o modelo de interceptadores, descrito no próximo capítulo, é capaz de ativar dinamicamente uma propriedade não-funcional.

A biblioteca de Lua fornece as funções responsáveis por realizar o carregamento e a ativação de código em tempo de execução. Dentre estas funções, temos a função *dofile()* que carrega o código Lua a partir de um arquivo e o executa. Um exemplo do uso da função *dofile()* é mostrado no código 2.1 abaixo. Como podemos ver na linha 3, a função *dofile()* recebe como parâmetro uma *string* contendo o nome do arquivo e o diretório em que está localizado.

Código 2.1: *Exemplo de utilização da função dofile()*

```
1  function Executar_Codigo_Dinamicamente()
2
3      dofile("/diretorio1/diretorio2/Arquivo_Carregado_Dinamicamente.lua")
4
5  end
6
7  -- A linguagem Lua não possui uma função principal como
8  -- a função main() da linguagem C. Para o código ser executado,
9  -- basta definir uma chamada explícita desta função como mostrado abaixo.
10
11 Executar_Codigo_Dinamicamente()
```

O arquivo com código Lua carregado pela função *dofile()* pode ser qualquer programa Lua, como aquele apresentado no código 2.2.

Código 2.2: *Arquivo carregado pela função dofile()*

```
1  -- Como a linguagem Lua não possui uma função main() como a
2  -- a linguagem C, instruções simples como uma chamada a função
3  -- print, que serve para imprimir strings na tela do terminal,
4  -- podem ser chamadas sem que haja necessidade de estarem definidas
5  -- dentro de uma função.
6
7  print("Código carregado dinamicamente\n")
```

Entretanto, caso o código no arquivo tenha algum erro, a função *dofile()* propaga esse erro, resultando na interrupção da aplicação principal. Um outra função, chamada *loadfile()*, também carrega um código Lua a partir de um arquivo da mesma forma que a função *dofile()* mas não o executa. No entanto a função *loadfile()* não propaga o erro, podendo este ser tratado dentro da aplicação principal, sem que haja interrupção. Outra

função, chamada *loadstring()*, é similar a *loadfile()*, exceto pelo fato de que carrega um código Lua a partir de uma *string*, e não de um arquivo, como o mostra o código 2.3.

Código 2.3: *Arquivo carregado pela função loadstring()*

```
1  funcao_anonima, mensagem_erro = loadstring("i = i + 1")
2
3  i = 0
4
5  if not funcao_anonima then
6     print("Erro no carregamento do código. Motivo: "..mensagem_erro.."\n")
7  else
8     funcao_anonima()
9     print(i.."\n")
10 end
```

Como pode ser observado, a função *loadstring()* retorna valores para duas variáveis. Se o carregamento do código tiver sido feito com sucesso, a variavél *funcao_anonima* recebe este código e a variável *mensagem_erro* recebe um valor nulo. Caso contrário, *funcao_anonima* recebe um valor nulo e *mensagem_erro* recebe uma *string* contendo a descrição do erro que surgiu no carregamento do código. A função *loadfile()* funciona da mesma forma, com a diferença de que o código está contido num arquivo. Quando um código em Lua é carregado através da função *loadstring()* ou *loadfile()* demonstradas anteriormente, a linguagem Lua insere este código dentro de uma função anônima que não possui parâmetros. Por exemplo, o trecho de código na forma de *string* mostrado em 2.3 (linha 1), inserido em uma função anônima por *loadstring()*, ficaria equivalente ao mostrado em 2.4.

Código 2.4: *Exemplo de um código equivalente ao retornado pela função loadstring() ou loadfile()*

```
1  function ()
2
3     i = i + 1
4
5  end
```

Então para chamar esta função anônima, basta utilizar a variavél *funcao_anonima*, que contém o código carregado como mostrado na linha 8 do código 2.3. No entanto, existem casos em que a aplicação deseja carregar um código Lua que possui uma função principal que precisa ser acessada diretamente, pois pode haver necessidade de transferir

um ou mais parâmetros e/ou receber algum resultado. Para que isto seja possível, o código a ser carregado deve retornar a sua própria função principal como mostrado no código 2.5 (linha 9).

Código 2.5: *Exemplo de um código que retorna sua função principal*

```
1  function funcao_principal(x, y)
2
3      local z = x + y
4
5      return z
6
7  end
8
9  return funcao_principal -- Retornando a função principal
10                         -- para poder ser acessada diretamente
11                         -- pela aplicação principal
```

Em seguida, o código acima deve ser inserido dentro de uma função anônima como mostrado em 2.6 através da função *loadstring()* ou *loadfile()*. Desta forma, quando a função anônima for chamada, será retornada esta função principal como mostrado no código 2.6 (linha 11).

Código 2.6: *Código definido acima retornado pela função loadstring() ou loadfile()*

```
1  function ()
2
3      function funcao_principal(x, y)
4
5          local z = x + y
6
7          return z
8
9      end
10
11     return funcao_principal -- Quando a função anônima for chamada,
12                             -- esta função retornará a função principal
13 end
```

Então, para acessar esta função, basta chamá-la usando o nome da variável que recebeu a função principal, podendo assim receber ou passar parâmetros. O código

definido em 2.7 mostra como é feito o carregamento e chamada desta função principal. Na linha 1, a função *loadfile()* carrega um arquivo chamado *Arquivo_Lua_1.lua*, que contém o código definido em 2.5, que é retornado dentro de uma função anônima para a variável *funcao_anonima*, caso não ocorra nenhum erro. Na linha 7, a função anônima é chamada através da variável *funcao_anonima* que retorna a função principal para a variável *funcao_principal*. A função principal é chamada utilizando a variável *funcao_principal* (linha 9) que recebe dois inteiros como parâmetros e retorna o valor da soma para a variável *resultado*.

Código 2.7: *Carregamento dinâmico de uma função que recebe parâmetros e/ou retorna algum resultado*

```
1  funcao_anonima, mensagem_erro = loadfile("/diretorio1/diretorio2/Arquivo_Lua_1.lua")
2
3  if not funcao_anonima then
4     print("Erro no carregamento do código. Motivo: "..mensagem_erro.."\n")
5
6  else
7     local funcao_principal = funcao_anonima()
8
9     local resultado = funcao_principal(2, 7)
10
11    print("O resultado é: "..resultado..".\n")
12
13 end
```

Outra forma de carregar estas funções dinamicamente é através da função *pcall()* como mostrado em 2.8. A função *pcall()* evita que a aplicação principal seja interrompida, se houver algum erro, e ainda retorna um mensagem descrevendo este erro. Como podemos observar, a função *pcall()* é chamada duas vezes. Na primeira vez, *pcall()* retorna a função principal do código para a variável *funcao_principal* (linha 7) caso não haja nenhum erro. Se houver algum, a variável *funcao_principal* recebe um valor nulo e *mensagem_erro* recebe uma *string* com a descrição do erro na chamada da função. Na segunda vez, *pcall()* chama a função principal contida na variável *funcao_principal* passando os dois inteiros como parâmetros. Em seguida, se não surgir algum erro, o valor é retornado pela função principal para a variável *resultado* (linha 13). No caso de ocorrer algum erro, a variável *ok* recebe um valor nulo e *resultado* recebe uma *string* contendo a descrição do erro.

Código 2.8: *Utilização da função pcall para chamar as funções carregadas dinamicamente*

```
1  funcao_anonima, mensagem_erro = loadfile("/diretorio1/diretorio2/Arquivo_Lua_1.lua")
2
3  if not funcao_anonima then
4     print("Erro no carregamento do código. Motivo: "..mensagem_erro.."\n")
```

```
5
6  else
7      local funcao_principal, mensagem_erro = pcall(funcao_anonima) -- OBTIÉM A FUNÇÃO
           PRINCIPAL
8
9      if not funcao_principal then
10         print("Erro na obtenção da função\n".."Motivo do erro: "..mensagem_erro.. "\n")
11
12     else
13         local ok, resultado = pcall(funcao_principal, 2, 7) -- CHAMA A FUNÇÃO PRINCIPAL
14
15         if not ok then
16               print("Erro na chamada da função \n".."Motivo do erro: "..resultado.. "\n")
17
18         else
19            print("O resultado é: "..resultado..".\n")
20
21         end
22
23     end
24
25
26
27 end
```

Outra característica de Lua, que é também importante para o modelo de interceptadores, é o fato de suas variáveis e tabelas receberem valores de qualquer tipo. Podemos, por exemplo, escrever o trecho de código <a = 2.3 >sem necessidade de definir o tipo da variável a. No caso das tabelas de Lua, que são vetores associativos, pode-se escrever um trecho de código <T[1] = 12.2 >sendo desnecessário definir o tipo da tabela T. Além disso, as tabelas de Lua também podem ser indexadas com qualquer tipo de valor.

Diante destes aspectos, podemos afirmar que Lua é uma linguagem tipada dinamicamente. Desta forma, uma variável ou tabela podem receber outro tipo de valor, tal como uma *string*, como por exemplo, <a = Ïvariavel Lua>ou <T[1] = Ïtabela Lua>. Portanto, uma mesma declaração de variável ou tabela podem ser utilizadas para receber diferentes tipos de dados retornados como resultado por funções chamadas em tempo de execução. Com este recurso da linguagem Lua, o modelo de interceptadores pode receber diferentes tipos de informação sobre um ambiente de execução utilizando a mesma definição de uma tabela ou variável.

2.3.3 *Middleware* Reflexivo

Plataformas de *middleware* reflexivo podem ser formadas a partir de plataformas de *middleware* convencionais, às quais são adicionados mecanismos reflexivos, tornando-as mais flexíveis, dinâmicas e configuráveis. Outra opção é a criação do *middleware* reflexivo a partir do zero, utilizando os princípios da reflexão computacional no seu desenvolvimento.

A configuração de um *middleware* reflexivo pode ser alterada dinamicamente pelas aplicações que o utilizam e também pelo próprio *middleware* para se adaptar às mudanças que ocorrem no ambiente distribuído. Por exemplo, uma aplicação multimídia pode melhorar seu desempenho reconfigurando o *middleware* reflexivo para utilizar um protocolo de comunicação mais adequado para a infra-estrutura de rede em uso, que pode ser a Internet, uma rede sem fio ou então uma rede local. Em um outro exemplo, um *middleware* reflexivo pode se reconfigurar para utilizar um algoritmo de encriptação para codificar e decodificar mensagens atendendo às atuais políticas de segurança de uma rede.

Estas reconfigurações podem ser feitas, por exemplo, através de *meta-objetos* que mantêm uma representação interna da configuração do *middleware* reflexivo. Portanto, quando o ambiente de execução sofre alguma mudança, os *meta-objetos* podem ser utilizados para verificar se a configuração atual do *middleware* precisa ser modificada para se adequar a essa alteração.

A seguir, serão apresentados dois casos de plataformas de *middleware* reflexivo. O primeiro é o *DynamicTAO* [33], que foi desenvolvido a partir de uma plataforma de *middleware* convencional chamada TAO. Já o segundo, chamado de *Open ORB* [5], foi desenvolvido desde o início com intuito de oferecer uma arquitetura para a construção de *middeware* baseado nos conceitos da reflexão computacional.

DynamicTAO

O *DynamicTAO* é a extensão de um ORB baseado em CORBA e escrito em C++ chamado TAO [3]. As extensões presentes no *DynamicTAO* possibilitam a reconfiguração, em tempo de execução, do seu estado interno e das aplicações que o utilizam. Este ORB reflexivo está inserido dentro do *2K*, um sistema operacional distribuído baseado em objetos CORBA [48]. O sistema *2K* é implementado como *middleware* sobre sistemas operacionais tais como Linux, Solaris e Windows. O *DynamicTAO* é utilizado pelo sistema *2K* como meio para fornecer suporte a reflexão computacional.

Em *DynamicTAO*, a reificação é feita através de uma coleção de entidades conhecidas como configuradores de componentes (*component configurators)* [29, 30]. Essas entidades servem para armazenar as dependências entre os componentes do ORB

e os componentes das aplicações, e também entre os componentes do próprio ORB. A Figura 2.8 apresenta o mecanismo de reificação do *DynamicTAO*.

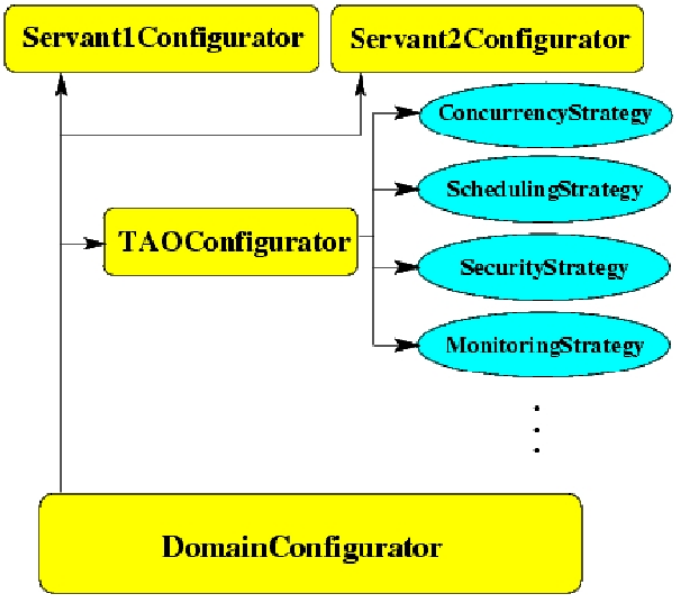

Figura 2.8: *Estrutura Interna do ORB DynamicTAO [33]*

Cada processo que executa no DynamicTAO contém uma instância de um configurador de componente chamado *DomainConfigurator*, que é responsável por manter referências para as instâncias do ORB e para os objetos servidores executando naquele processo.

Como mostrado na figura acima, o *TAOConfigurator* é o configurador de componente responsável por permitir a reconfiguração dinâmica dos componentes do *DynamicTAO* que implementam o controle de concorrência, escalonamento, segurança e monitoramento.

Além disso, *DynamicTAO* oferece uma meta-interface que pode ser utilizada por programadores para a depuração e testes, por administradores de sistemas para manutenção e configuração, ou por outros componentes de software, que podem inspecionar e reconfigurar as camadas internas do ORB baseados em informações coletadas de outras fontes, tais como monitores de utilização de recursos.

Existe também uma outra meta-interface similar, sendo que seu uso é direcionado para agentes móveis [32]. Neste caso, os agentes móveis são criados por administradores de sistemas e injetados na rede. Então estes agentes vão se deslocando de um ORB para outro, monitorando e reconfigurando cada nó de acordo com as instruções dadas pelo administrador.

A reflexão estrutural é implementada através dos configuradores de componentes, que auxiliam na reificação da estrutura do sistema. Reflexão comportamental é obtida através de interceptadores e do uso do DSRT para o gerenciamento de recursos.

Interceptadores podem ser instalados e carregados dinamicamente, tanto no lado do cliente quanto no lado do servidor. Geralmente, são utilizados para adicionar propriedades não-funcionais tais como criptografia, compressão e qualidade de serviço, entre outros. No caso do *DynamicTAO*, interceptadores foram utilizados para implementar o monitoramento de chamadas a objetos distribuídos e para implementar um sofisticado sistema de segurança baseado em capacidades ativas [33].

No sistema *2K*, o componente responsável pelo gerenciamento de recursos é o DSRT (*Dynamic Soft Real-Time Scheduler*) [40]. No entanto, este escalonador não faz parte do sistema. Por isso, para poder ser utilizado, o DSRT deve ser carregado dinamicamente pelo sistema *2K*. Depois de ser carregado, o DSRT atua como um processo de nível de usuário em sistemas operacionais tais como Windows, Linux e Solaris. Através das *interfaces* de tempo real de baixo nível oferecidas por esses sistemas, o DSRT realiza controle de admissão, negociação de recursos, reserva de recursos e escalonamento em tempo real. Por estar presente dentro do sistema *2K*, o *DynamicTAO* também pode acessar o DSRT para obter informações sobre o gerenciamento de recursos.

Open ORB

O projeto *Open ORB* tem por objetivo desenvolver plataformas de *middleware* reconfiguráveis dinamicamente que forneçam suporte para aplicações com requisitos dinâmicos, incluindo aquelas que envolvem multimídia distribuída e mobilidade [12].

A arquitetura do *Open ORB* é baseada em meta-níveis, ou seja, possui um nível base e um meta-nível. Enquanto o nível base é composto pelos componentes que implementam as funcionalidades do *middleware*, o meta-nível é constituído por mecanismos que mantêm uma representação interna do conjunto de componentes da plataforma e permite sua inspeção e adaptação dinâmica.

Para lidar com a elevada complexidade que é comum nas arquiteturas reflexivas para *middleware*, o meta-nível é então dividido em meta-espaços. Cada meta-espaço é constituído por um grupo de meta-objetos responsáveis pela representação interna de um componente específico do *middleware*.

O Open ORB define quatro modelos de meta-espaço, que são agrupados de acordo com a distinção entre reflexão estrutural e comportamental. Estes modelos são ilustrados na Figura 2.9.

A reflexão estrutural é representada por modelos de meta-espaço denominados de *Interfaces* e *Architeture* sendo que os outros dois modelos, chamados de *Interception* e *Resources*, são dedicados à reflexão comportamental.

O modelo *Interfaces* está relacionado com a representação externa de um componente, em termos dos conjuntos de *interfaces* fornecidas e requisitadas. O protocolo de meta-objetos (MOP) associado a este modelo oferece meios para enumerar e procurar os

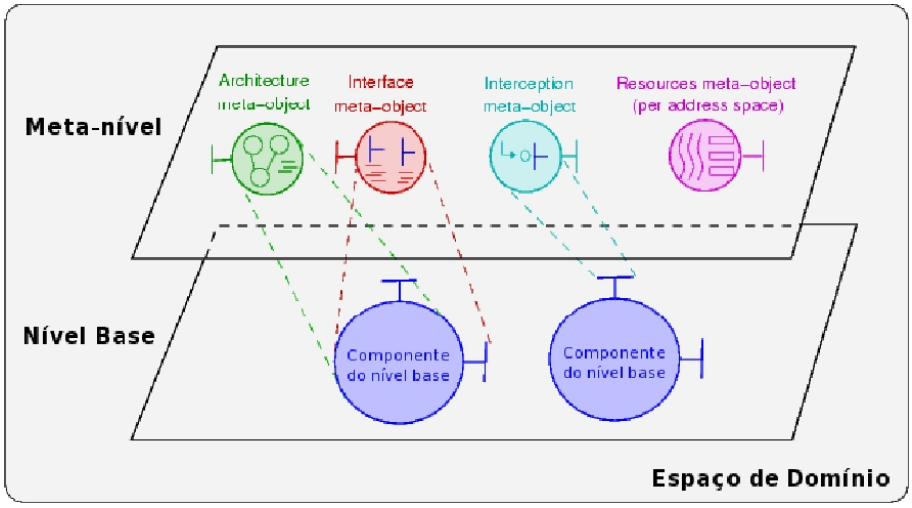

Figura 2.9: *Estrutura do meta-espaço em Open ORB [5]*

elementos que definem a *interface* do componente, permitindo, por exemplo, a descoberta dinâmica de serviços fornecidos por um componente.

Já o modelo *Architeture* está relacionado com a representação da implementação interna de um componente maior, formado a partir de componentes mais simples, em termos de sua arquitetura de software. Esta auto-representação consiste de duas partes: um grafo de componentes representando as interconexões entre os componentes mais primitivos que por sua vez constituem um componente maior reificado, e um conjunto de restrições arquiteturais definindo as regras necessárias para validar as configurações desses componentes. O MOP associado a este modelo auxilia na inspeção e adaptação da arquitetura do software, por exemplo, para adicionar, remover ou substituir componentes e mudar restrições, tornando possível a adaptação dinâmica.

O modelo *Interception*, com o uso de seu MOP, habilita a manipulação de propriedades não-funcionais sob a forma de interceptadores que realizam pré e pós-processamento das interações emitidas e recebidas em uma *interface*.

O último modelo, *Resources*, oferece acesso estruturado aos recursos subjacentes das plataformas e o gerenciamento destes recursos. O MOP permite a inspeção e reconfiguração dos recursos alocados para atividades particulares no sistema através da adição ou retirada de recursos ou alterando os parâmetros e algoritmos necessários para o gerenciamento de recursos.

2.4 *Middleware* de Grade Reflexivo

Plataformas de *middleware* de grade também podem se beneficiar da reflexão computacional para se adequarem às freqüentes alterações presentes nas grades computacionais. Assim como um *middleware* reflexivo, uma plataforma de *middleware* de grade

reflexiva pode ser desenvolvida utilizando os princípios da reflexão computacional desde o início. Outra opção é a criação do *middleware* de grade reflexivo a partir de um *middleware* de grade estático através da inserção de novos componentes com capacidades reflexivas ou então pela inclusão de mecanismos reflexivos nos componentes já existentes.

Esta última opção é mais conveniente para os desenvolvedores que já criaram um *middleware* de grade que ainda não possui suporte a adaptação dinâmica. A adição de novos componentes com recursos adaptativos certamente fará com que um *middleware* de grade seja dinâmico. No entanto, seria necessário modificar a sua implementação e, conseqüentemente, a sua arquitetura.

A seguir, serão apresentados dois trabalhos que têm por objetivo fornecer mecanismos reflexivos para auxiliar as plataformas de *middleware* de grade a lidar com o dinamismo das grades computacionais. Além disso, esses trabalhos incluem também aspectos autonômicos, ou seja, permitem que um *middleware* de grade realize adaptações sem a necessidade de configuração por parte do usuário.

O primeiro é o AutoGrid, um *middleware* de grade reflexivo e autônomo, que utiliza o InteGrade como base para sua implementação. Assim, o InteGrade foi alterado através da inserção de mecanismos autonômicos e de reflexão em sua arquitetura para ter condições de oferecer recursos de adaptação dinâmica. O segundo trabalho é o AutoMate, um arcabouço que busca tornar um *middleware* de grade baseado no padrão OGSA, tal como o Globus, capaz de prover suporte a aplicações autônomas com recursos de adaptação dinâmica.

2.4.1 AutoGrid

O AutoGrid [51] é um *middleware* de grade auto-gerenciável e autonômico. Na sua implementação, o InteGrade foi utilizado como base, ao qual foram incorporados mecanismos de adaptação dinâmica para torná-lo capaz de lidar com o alto dinamismo presente nas grades computacionais.

Arquitetura do AutoGrid

A arquitetura do AutoGrid reutiliza vários componentes do InteGrade e também inclui outros, com características autonômicas. Os componentes do InteGrade que foram reutilizados são o *Application Submission and Control Tool (ASCT)*, o *Application Repository (AR)*, o *Local Resource Manager (LRM)*, o *Global Resource Manager (GRM)* e o *Execution Manager (EM)*. Os demais componentes da arquitetura do AutoGrid são apresentados abaixo [51]:

- *Monitoring Service (MS)*: coleta informações sobre o estado dos recursos dos nós da grade como, por exemplo, memória, CPU, disco rígido, uso da rede e

aplicações. Cada recurso é composto por uma ou mais propriedades, que são atributos monitoráveis, tais como memória disponível, nível de utilização da CPU, largura de banda e latência da rede, e quantidade de *threads* da aplicação.

- *Local Event Service (LES)*: recebe notificações dos MSs distribuídos pelos nós da grade e notifica os componentes registrados quando mudanças na disponibilidade de um recurso específico são detectadas.
- *Event Process System (EPS)*: um serviço de eventos distribuídos que detecta eventos gerados por diferentes nós da grade.
- *Dynamic Reconfiguration System (DyReS)*: mecanismo de adaptação que aplica ações de reconfiguração na aplicação em resposta a mudanças no ambiente de execução. O *DyRes* também coordena os outros componentes envolvidos no processo de reconfiguração.
- *Stable Storage*: um repositório de dados distribuído que armazena o estado das aplicações (*checkpoints*).

A Capacidade de Auto-Gerenciamento do AutoGrid

Para poder ser auto-gerenciável, o componentes do AutoGrid foram implementados através do arcabouço *Adapta* [50], utilizado para o desenvolvimento de aplicações auto-adaptativas que separam, de forma clara, o código responsável pelas regras de negócio do código que define os mecanismos de adaptação. A arquitetura deste arcabouço é baseada em reflexão computacional, onde *Adapta* é composta de um nível base, responsável pela lógica da aplicação e por um meta-nível, responsável pelo processamento reflexivo. Portanto, todo componente baseado no arcabouço *Adapta* é flexível e reconfigurável

Aspectos Autonômicos do AutoGrid

O *middleware* de grade AutoGrid possui quatro aspectos autonômicos, que são descritos abaixo [51]. Em seguida, são descritos os mecanismos do AutoGrid que implementam cada uma destas propriedades.

- *Domínio de Contexto*: o sistema deve ter conhecimento do seu ambiente de execução. Desta forma, é possível para o sistema reagir de forma adequada às mudanças.
- *Auto-Configuração*: o sistema deve oferecer suporte para ações de reconfiguração com base no estado de seu ambiente de execução.
- *Auto-Recuperação*: o sistema deve estar ciente das possíveis falhas que podem ocorrer no seu funcionamento. Desta forma, o sistema pode se reconfigurar para continuar a operar normalmente e então se recuperar através técnicas dinâmicas para tratamento de falhas.

- *Auto-Otimização*: o sistema deve ter condições de detectar quedas no seu desempenho e tomar medidas para evitá-las.

Mecanismo de Domínio de Contexto

Este mecanismo permite ao AutoGrid observar o seu ambiente de execução, obtendo informações individuais sobre os nós da grade, tais como disponibilidade de CPU e memória utilizada, além de informações gerais sobre a grade, como, taxa de requisições para execução de aplicações e a ocorrência de falhas. Essas informações são obtidas através do *Monitoring Service (MS)*, que inspeciona o hardware subjacente e o ambiente de execução em cada nó da grade usando objetos monitores. Cada objeto é configurado para detectar mudanças na disponibilidade de um recurso específico, como, a largura de banda livre na rede. Monitores podem ser dinamicamente instanciados para introduzir novos requisitos de monitoramento ou substituídos em tempo de execução para se adequar à diversidade das plataformas computacionais.

Quando um certo recurso se torna disponível, o *MS* notifica o *Local Event Service (LES)*. Por sua vez, o LES verifica se a condição de disponibilidade de recursos foi atendida. Essa condição, por exemplo, pode ser o nível de tinta de uma impressora ou uma quantidade de espaço livre no disco rígido. A avaliação dessa condição é feita de forma a minimizar a quantidade de mensagens enviadas para os nós e é baseada em uma expressão booleana fornecida pelo desenvolvedor do *middleware* de grade como parte da definição do evento. Assim, para lançar a notificação de um evento, essa expressão booleana deve se manter *verdadeira* durante um período de tempo definido pelo usuário. Esta abordagem evita que notificações sejam geradas quando surgem situações temporárias, tais como um aumento inesperado no uso de um recurso, como a CPU, por causa de programa com um alto custo de processamento que acabou de ser iniciado.

Logo que o evento é detectado, o LES notifica o *Event Processing System (EPS)* e todos os componentes registrados que estão associados a uma instância do *DyReS*. O EPS é requisitado sempre que a decisão de reconfigurar os componentes do AutoGrid deve levar em conta a combinação de eventos detectados em diferentes nós da grade. Por exemplo, algoritmos dinâmicos para balanceamento de carga definidos com base em migrações de aplicação têm que considerar a utilização da CPU em cada nó e também as condições da rede. Ao detectar um evento distribuído, o EPS notifica os eventos aos componentes registrados. Então o DyReS pode decidir se deve ou não reconfigurar esses componentes.

Mecanismo de Auto-Configuração

Cada componente adaptativo da arquitetura do AutoGrid está associado a uma instância do DyReS, que corresponde ao seu meta-nível. O DyReS recebe notificações de

eventos tanto do LES quanto do EPS. Através dessas notificações, o DyReS verifica se precisa reconfigurar o componente. Existem dois tipos de reconfiguração que o DyReS pode realizar:

- *Mudança de Parâmetros da Aplicação*: o mecanismo de atualização de parâmetros utiliza a abordagem de método *callback*. O desenvolvedor do *middleware* de grade introduz, durante o projeto, métodos *callback* para serem invocados em cada classe que possui um parâmetro atualizável. AutoGrid obtém a referência do *callback* e o invoca dinamicamente usando novos valores informados pelo processo de reconfiguração. Como exemplo, a atualização dos parâmetros pode ser requisitada para modificar o intervalo entre os *checkpoints*.
- *Substituição de Algoritmos da Aplicação*: o processo de substituição de algoritmos necessita de um *proxy* que introduza uma camada extra acima dos objetos que serão substituídos, mantendo a adaptação transparente para os clientes. O *proxy* também gerencia a transferência de estado durante o processo de substituição. O estado consiste de um conjunto de variáveis que representa a situação atual da execução do algoritmo. Já que as variáveis são, na verdade, atributos da classe, o *proxy* pode observar o objeto que usa o algoritmo, armazenar seu estado, e carregá-lo dentro de outro objeto. A substituição de algoritmo pode ser necessária, por exemplo, para mudar o algoritmo de escalonamento atual da grade por um outro que seja mais adequado ao ambiente de execução.

Além destes dois tipos de reconfiguração, o DyReS permite que os usuários tenham condições de inserir outros métodos de reconfiguração, tais como adição, remoção ou substituição de componentes.

Mecanismo de Auto-Recuperação

As grades computacionais são bastante propensas a falhas por causa de diversos fatores, tais como sua natureza dinâmica e autônoma. Para assegurar a continuidade da execução das aplicações, diversas técnicas de tratamento de falhas podem ser aplicadas, incluindo:

- *Reinicialização*: recomeça uma aplicação do zero.
- *Replicação*: quando uma aplicação da grade começa sua execução, uma determinada quantidade de cópias de uma aplicação, conhecidas como réplicas, é criada. Se a aplicação original falhar, uma de suas réplicas a substitui de forma transparente para o usuário. Quando a aplicação terminar, o *middleware* de grade deve descartar as outras réplicas e retornar os resultados para o usuário.

- *Checkpointing*: armazena periodicamente o estado da aplicação num repositório durante um intervalo em que não tenha ocorrido qualquer falha. No caso de acontecer uma falha, a aplicação reinicia a partir do último ponto salvo.

Dentro das grades computacionais, a técnica para tratamento de falhas mais adequada depende de diversos fatores relacionados ao ambiente de execução, tais como o MTBF [15]. O mecanismo de auto-recuperação do AutoGrid pode selecionar automaticamente a técnica para tratamento de falhas mais apropriada, de acordo com os fatores apresentados e também ajustar o intervalo entre *checkpoints* consecutivos, baseado apenas na carga computacional.

O AutoGrid pode decidir dinamicamente a quantidade de réplicas a serem geradas para uma determinada aplicação com base no MTBF do ambiente de execução e também no MTBF da própria aplicação. O GRM é o componente responsável por gerenciar o mecanismo de tolerância a falhas. Para isso, o GRM foi associado a um DyReS que modifica dinamicamente o valor do MTBF e recalcula o número de réplicas que devem existir. O algoritmo utilizado para este propósito tenta manter o período de tempo em que uma aplicação executa no AutoGrid, sem ocorrer nenhuma falha, o mais próximo possível ao valor definido no MTBF desta aplicação usando o mínimo de réplicas possível.

Mecanismo de Auto-Otimização

O dinamismo presente nas grades computacionais é melhor observado nas grades oportunísticas que aproveitam poder computacional ocioso das redes para a execução de aplicações paralelas intensivas. Já que os recursos dos nós de uma grade não são dedicados como em um *cluster*, geralmente as aplicações que executam nesses ambientes têm um desempenho mais baixo. Para contornar este problema, existem técnicas para otimização do desempenho das aplicações, tais como balanceamento de carga, escalonamento adaptativo e re-escalonamento dinâmico, entre outras.

O mecanismo de auto-otimização do AutoGrid melhora o desempenho da aplicação que executa na grade através da substituição dos algoritmos de escalonamento em tempo de execução, de acordo com o estado do ambiente de execução da grade. Por exemplo, o algoritmo de escalonamento *Minimum Completion Time* (MCT) é mais indicado quando a taxa de requisição de tarefas está acima de um determinado limite. No entanto, um outro fator presente no ambiente de execução das grades que afeta a seleção do algoritmo de escalonamento mais apropriado é a utização de recursos. Neste caso, a heurística Máximo-Mínimo poderia ser utilizada para maximizar a concorrência de tarefas quando a utilização de recursos estiver baixa. Durante a substituição do algoritmo, o AutoGrid gerencia a transferência de estados, que leva em conta a fila de aplicações armazenada

pelo algoritmo. Como o GRM é o responsável por substituir o algoritmo de escalonamento, o mecanismo de auto-otimização fica localizado dentro deste componente.

2.4.2 AutoMate

O AutoMate é um arcabouço que permite o desenvolvimento de aplicações de grade autônomas, que tenham ciência de seu ambiente de execução e sejam capazes de realizar auto-configuração, auto-composição, auto-otimização e adaptação [1]. Especificamente, o AutoMate oferece meios para o desenvolvimento de aplicações de grade autonômicas através da composição dinâmica de componentes autônomos e também abordagens para que os sistemas de *middleware* de grade ofereçam suporte para esse tipo de aplicação. A visão geral do projeto AutoMate é mostrada na Figura 2.10. Os objetivos principais desse projeto são [1]:

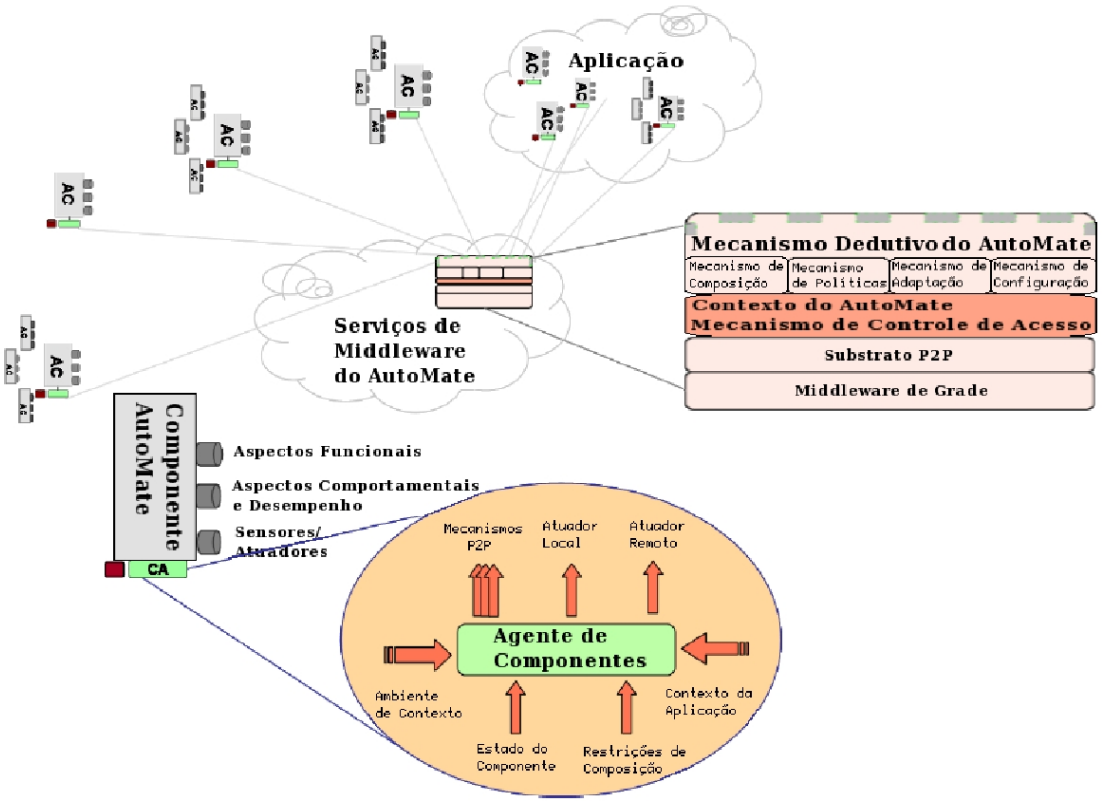

Figura 2.10: *Visão Geral do AutoMate [1]*

- *Definição de Componentes Autônomos*: componentes autônomos devem fornecer informações sobre seus aspectos funcionais, operacionais e de controle. Estes aspectos fornecem informações sobre o comportamento do componente, requisitos de recursos, desempenho, interatividade e adaptabilidade. Além disso, estes aspectos encapsulam também sensores e gerenciadores que controlam as políticas de acesso.

Através destes aspectos, os componentes autônomos podem se configurar, gerenciar, adaptar e otimizar sua execução.

- *Composição Dinâmica de Aplicações Autônomas*: criação de aplicações autônomas a partir de componentes também autônomos através do uso de mecanismos e infra-estrutura de suporte providos pelo AutoMate. O desenvolvimento destas aplicações é baseada em políticas e restrições definidas e ativadas em tempo de execução, levando em conta os recursos (aplicações, serviços, armazenamento de dados) e componentes da grade, requisitos e capacidades.
- *Serviços de Middleware Autônomos*: O AutoMate possui a capacidade de estender um *middleware* de grade com um substrato *peer-to-peer*, serviços cientes de contexto, mecanismos de dedução *peer-to-peer* para composição, configuração, e gerenciamento de aplicações autônomas. A utilização de redes *peer-to-peer* serve para que os componentes e recursos interajam entre si, possibilitando que as aplicações se comportem de forma autônoma.

Arquitetura do AutoMate

A arquitetura do AutoMate é mostrada na Figura 2.11. Como podemos observar na figura, o AutoMate utiliza um *middleware* de grade baseado no padrão OGSA [43]. A seguir, são descritos os componentes desta arquitetura.

- *Camada de Sistema*: esta camada utiliza um *middleware* de grade como infra-estrutura e estende os serviços definidos no padrão OGSA (segurança, gerenciamento e informação de recursos, gerenciamento de dados) para fornecer suporte para comportamento autônomo. Além disso, esta camada oferece serviços especializados, tais como semântica de mensagens *peer-to-peer*, eventos e notificação.
- *Camada de Componentes*: esta camada é responsável pela definição, execução e gerenciamento, em tempo de execução, de componentes autônomos. A camada é formada por sub-camadas com mecanismos de auto-configuração, adaptação e otimização, além de serviços como descoberta e contexto, entre outros.
- *Camada de Aplicação*: esta camada utiliza os serviços das camadas de sistema e de componentes para prover suporte a composição autônoma e interações dinâmicas entre componentes.
- *Mecanismos do AutoMate*: são redes (*peer-to-peer*) de agentes descentralizados no sistema. O *Mecanismo Ciente de Contexto* é composto de agentes e serviços que fornecem informações de contexto em diferentes níveis para ativar comportamentos autônomos. O *Mecanismo Dedutivo* é composto de agentes que são parte das aplicações, componentes, serviços e recursos, oferecendo a capacidade de tomar

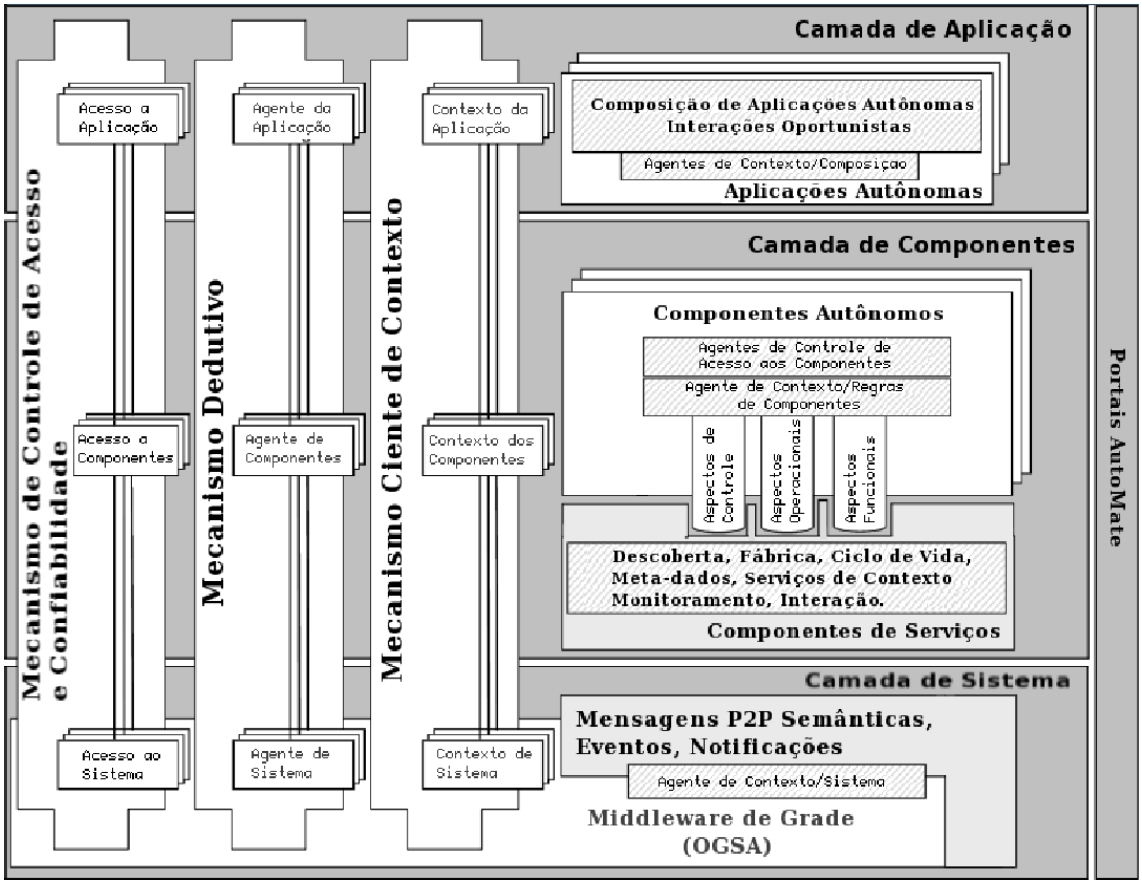

Figura 2.11: *Arquitetura do AutoMate [1]*

decisões coletivas para habilitar o comportamento autônomo. Finalmente, o *Mecanismo de Controle de Acesso e Confiabilidade* é composto de agentes de controle de acesso e provê controle dinâmico ciente de contexto para todas as interações no sistema.

2.5 Considerações Finais

Neste capítulo, foram apresentados e contextualizados os principais fundamentos que representam a base para este trabalho. O modelo de interceptadores dinâmicos definido neste trabalho (descrito no Capítulo 3) foi aplicado no ORB OiL [38] que é uma das plataformas de *middleware* convencionais utilizada pelo InteGrade. No início do capítulo, uma descrição geral deste tipo de plataforma de *middleware* foi apresentada.

Em seguida, os principais conceitos de *middleware* de grade foram mostrados, juntamente com a demonstração de alguns trabalhos nesta área, incluindo o InteGrade. Depois, foram apresentados os principais conceitos de reflexão computacional, uma das abordagens mais utilizadas pelas plataformas de *middleware* para lidar com o dinamismo existente nos ambientes computacionais distribuídos. A reflexão computacional foi apli-

cada primeiramente em linguagens de programação, sendo então empregada em outras áreas inclusive em sistemas distribuídos. Lua [27] é um exemplo deste tipo de linguagem que possui características reflexivas. Esta também foi a linguagem utilizada para implementar o ORB OiL. Neste capítulo, vimos alguns aspectos da linguagem Lua que foram bastante úteis para a implementação do modelo de interceptadores dinâmicos no OiL.

Como foi discutido, uma plataforma de *middleware* de grade reflexiva pode ser criada a partir de um *middleware* de grade que não possui suporte para adaptação dinâmica. Um dos modos de tornar um *middleware* reflexivo é através da adição de novos componentes com capacidades reflexivas em sua estrutura. O outro modo consiste em inserir mecanismos reflexivos nos componentes já existentes do *middleware* de grade, evitando assim a alteração de sua arquitetura. Desta forma, ao inserir interceptadores, que são um mecanismo reflexivo, no ORB OiL, os mesmos podem ser utilizados para permitir que um *middleware* de grade estático, como o InteGrade consiga modificar dinamicamente o comportamento de suas funcionalidades sem a necessidade de alterações na arquitetura e implementação.

A reflexão computacional se apresenta como um bom meio para permitir que diversos tipos de plataformas *middleware* tenham condições de lidar com o alto dinamismo cada vez mais comum nos ambientes distribuídos. Através da utilização destes conceitos de reflexão, uma plataforma de *middleware* obtém a capacidade de modificar a sua estrutura e o seu comportamento para sempre oferecer um suporte mais otimizado para as aplicações que necessitem de seus serviços. As plataformas de *middleware* reflexivo *DynamicTAO* e *Open ORB*, apresentadas neste capítulo, são um bom exemplo dos benefícios da reflexão computacional para as plataformas de *middleware*.

Entretanto apenas o uso da reflexão computacional não é suficiente para resolver todos os problemas relacionados com o alto dinamismo presente nos ambientes computacionais distribuídos. Por isso, o uso de outras abordagens, em conjunto com a reflexão computacional, é necessário para que as plataformas de *middleware* consigam se adaptar melhor às variações que ocorrem dentro de um ambiente distribuído. Um bom exemplo disso é a utilização de mecanismos autonômicos em plataformas de *middleware* junto com recursos de reflexão computacional. O AutoGrid e o AutoMate são dois trabalhos que possibilitam a um *middleware* de grade a capacidade de se adaptarem de forma autônoma, ou seja, sem a necessidade da intervenção do usuário.

Diante do que foi discutido, podemos concluir que reflexão computacional se mostra como uma boa opção para ajudar as plataformas de *middleware* de grade a se adaptarem às variações ocorridas em uma grade computacional. No entanto, reflexão é apenas uma solução parcial para o problema. Desta forma, o uso de outras abordagens, como por exemplo o uso de mecanismos de autonômicos, é necessário.

CAPÍTULO 3

O Modelo de Interceptadores Dinâmicos

Este capítulo apresenta a arquitetura e implementação do modelo de interceptadores dinâmicos desenvolvido com o intuito de introduzir mecanismos de adaptação dinâmica no InteGrade. Neste trabalho, o modelo de interceptadores dinâmicos foi implementado no OiL, que é um dos ORBs de comunicação utilizados pelo InteGrade.

O InteGrade utiliza o OiL para permitir que a comunicação entre seus componentes seja feita de forma transparente, sem se preocupar com questões tais como heterogeneidade ou protocolos de acesso. Através do modelo de interceptadores dinâmicos, o InteGrade também poderá utilizar o OiL para se adaptar às variações do ambiente de execução das grades computacionais.

Também, são apresentadas comparações do modelo de interceptadores com os sistemas Automate e o AutoGrid, descritos no Capítulo 2, e que também visam fornecer mecanismos de adaptação dinâmica para plataformas de *middleware* de grade. Por último, são apresentadas algumas considerações sobre a possibilidade de implementação do modelo de interceptadores no JacORB e também na versão do OiL baseada em componentes.

3.1 Conceitos básicos de Interceptadores

Interceptadores são mecanismos que estendem as funcionalidades básicas de um *middleware*, modificando o seu comportamento no que diz respeito a propriedades não-funcionais. Assim, interceptadores podem ser utilizados para inserção e configuração de propriedades não-funcionais, tais como tolerância a falhas, qualidade de serviço e compressão em plataformas de *middleware* de forma transparente, sem alterar sua arquitetura. Para ativar interceptadores dentro de um ORB, existem duas formas:

- Estática: neste caso, a decisão sobre a utilização de interceptadores é feita em tempo de compilação e antes de sua inicialização. Os interceptadores definidos na especificação CORBA [48], por exemplo, podem ser ativados estaticamente.

- Dinâmica: a decisão é feita em tempo de execução, podendo ser realizada antes ou depois da inicialização do ORB. Os interceptadores definidos em CORBA também podem ser ativados dinamicamente, sendo que a necessidade de seu uso deve ser decidida antes do início da execução do ORB.

Os interceptadores de CORBA também podem ser utilizados para ativar outros interceptadores de forma dinâmica como mostrado em [39]. Nesse trabalho, um arcabouço chamado ACT fornece meios para inserir mecanismos de adaptação dinâmica em ORBs baseados em CORBA em tempo de execução. Esse arcabouço é formado por dois componentes. O primeiro componente é um interceptador de CORBA, que é ativado estaticamente. O segundo componente é o núcleo do ACT. Este núcleo é o responsável por ativar interceptadores em tempo de execução. Desta forma, o interceptador de CORBA, ao ser inicializado, executará o núcleo do ACT. Por sua vez, este componente ativará um outro interceptador dinamicamente.

3.2 Visão Geral do Modelo de Interceptadores Dinâmicos

O InteGrade é um *middleware* de grade oportunista que serve para fornecer acesso a recursos ociosos de uma ou mais redes de computadores, de forma transparente, para a execução de aplicações. Entretanto, o InteGrade é ainda um *middleware* de grade estático, ou seja, não é capaz de se reconfigurar dinamicamente para lidar com as freqüentes alterações presentes nas grades computacionais, tais como disponibilidade de recursos e conectividade da rede.

Neste trabalho, um modelo de interceptadores dinâmicos foi desenvolvido para verificar a factibilidade e os benefícios de se utilizar mecanismos de adaptação dinâmica de um ORB de comunicação parar tornar um *middleware* de grade adaptativo.

Assim, este modelo de interceptadores foi inserido no ORB OiL [38], um dos ORBs de comunicação utilizado pelo InteGrade. Entre as suas principais características, este modelo de interceptadores possui a capacidade de detectar mudanças no ambiente de execução e no estado interno de um ORB através de um componente chamado *Monitor*, que será mostrado mais à frente. A utilização deste modelo de interceptadores permite que o ORB seja capaz de otimizar dinamicamente sua configuração sempre que houver variações no seu ambiente de execução e também em seu estado interno.

Desta forma, os mecanismos de adaptação do OiL contribuirão para que o InteGrade tenha a capacidade de observar o ambiente de execução das grades computacionais e adaptar sua configuração caso haja alguma mudança.

No entanto, os componentes do InteGrade, tais como o GRM ou EM, que utilizam o JacORB, não poderão usufruir dos funcionalidades oferecidas pelo modelo de

interceptadores. Isto ocorre devido ao fato de que o modelo de interceptadores ainda não foi implementado no JacORB. Portanto, apenas os componentes que utilizam o OiL, como por exemplo o LRM, aproveitarão, de forma mais direta, as funcionalidades oferecidas pelo modelo de interceptadores. Por isso, para que o InteGrade possa aproveitar totalmente as funcionalidades do modelo de interceptadores dinâmicos, este mecanismo de adaptação dinâmica deve ser inserido também no JacORB.

A vantagem obtida pelo InteGrade com o uso deste modelo de interceptadores dinâmicos é a introdução da capacidade de adaptação dinâmica com um mínimo de intervenção em sua arquitetura. Outra vantagem é a possibilidade de adicionar ou remover propriedades não-funcionais sem a necessidade de recompilar o InteGrade, ou mesmo os seus ORBs de comunicação.

Interceptadores são mecanismos associados à reflexão comportamental. Portanto, interceptadores não são capazes de alterar a estrutura interna do *middleware*, ou seja, normalmente não podem ser utilizados para monitorar e modificar a funcionalidade do *middleware*, ficando restritos a aspectos não-funcionais.

A Figura 3.1 mostra uma visão geral da abordagem seguida neste trabalho. Inicialmente, vemos o InteGrade como um *middleware* de grade estático que utiliza dois ORBs estáticos (JacOrb e OiL) como meios de comunicação entre seus componentes internos. Em seguida, o modelo de interceptadores é inserido dentro dos ORBs. Então, utilizando este mecanismo de adaptação dinâmica, estes ORBs serão capazes de manipular suas propriedades não-funcionais para se adequarem às modificações que ocorrem no seu ambiente de execução. Assim, o InteGrade, se beneficiando dos mecanismos de adaptação dinâmica proporcionados pelos seus ORBs, passará também a ser um *middleware* com capacidade de se adaptar dinamicamente.

Através do modelo de interceptadores dinâmicos, o InteGrade poderá modificar dinamicamente a maneira como seus componentes interagem entre si. Isto significa que o InteGrade poderá se adequar, em tempo de execução, às mudanças dos requisitos não-funcionais de seus componentes. Por exemplo, interceptadores poderiam criar réplicas de uma aplicação em execução para prover tolerância a falhas ou então criptografar as mensagens trocadas entre os seus componentes para oferecer maior segurança durante a comunicação pela rede.

No entanto, o modelo de interceptadores dinâmicos não tornará o InteGrade capaz de inserir, modificar ou remover componentes de sua arquitetura para se adequar às variações do ambiente de execução. Assim, o InteGrade não poderá, por exemplo, alterar o componente responsável pelo escalonamento de tarefas utilizando interceptadores.

Como já foi citado, o modelo de interceptadores definido neste trabalho foi implementado apenas no OiL. Por isso, o próximo passo será a implementação deste modelo de interceptadores dinâmicos no JacORB, de acordo com a descrição da abordagem

apresentada acima. No entanto, esta etapa deverá ser feita num próximo trabalho.

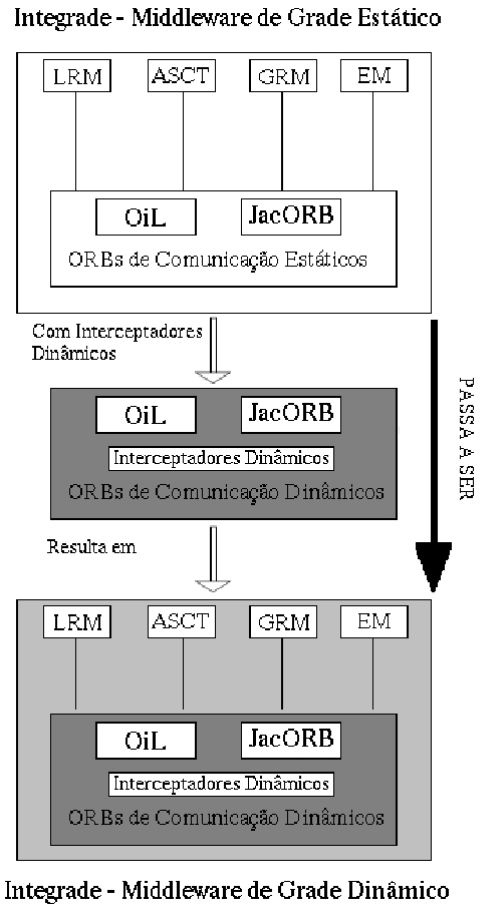

Figura 3.1: *Visão geral do projeto*

3.3 Arquitetura

A arquitetura do modelo de interceptadores dinâmicos foi definida através da utilização do padrão de projeto *Interceptor* [52], sendo composta por três componentes. O primeiro componente é o *Monitor*, responsável por inspecionar o ambiente de execução e o estado interno do ORB. Caso ocorra alguma mudança, tanto no ambiente de execução quanto dentro do ORB, o *Monitor* decidirá qual propriedade não-funcional deve ser adaptada. Quando o *Monitor* é chamado, um componente denominado *Context*, que será descrito mais à frente, deve ser passado como parâmetro. O *Monitor*, por sua vez, deve repassar este componente para o *Implementor*.

O segundo componente é o *Implementor*, que contém a implementação de uma propriedade não-funcional que será aplicada em resposta a alguma mudança detectada pelo *Monitor*.

O terceiro componente é o *Context*, que permite ao *Monitor* acessar as informações internas do ORB. A criação deste componente ocorre quando um ponto de interceptação inserido no ORB é atingido. O tipo de informação obtida pelo *Context* depende do evento que levou o ORB a invocar o *Monitor* neste ponto. Este evento pode ser, por exemplo, uma requisição do cliente sendo enviada para um objeto remoto ou então uma resposta vinda deste mesmo objeto remoto. Assim, o *Monitor* teria informações sobre as requisições e/ou respostas que foram enviadas e recebidas. Estas informações também podem ser utilizadas pelos componentes do tipo *Implementor* para a aplicação de uma propriedade não-funcional.

A Figura 3.2 apresenta a arquitetura do modelo de interceptadores dinâmicos. O *Monitor* pode ter referência a apenas um ou então vários componentes do tipo *Implementor*, cada componente *Implementor* implementa uma propriedade não-funcional. O *Context* fornece informações do estado interno ORB no ponto em que o interceptador está inserido, que é acessado pelo *Monitor* e então repassado ao *Implementor*. Quando ocorre uma alteração dentro do ORB ou no ambiente de execução, o *Monitor* deve carregar um dos componentes do tipo *Implementor* para lidar adequadamente com essa variação.

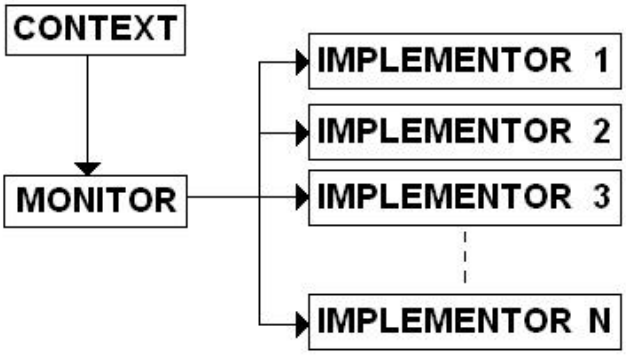

Figura 3.2: *Arquitetura do Modelo de Interceptadores Dinâmicos*

Uma instância do modelo de interceptadores pode ser inserida em diferentes pontos dentro de um ORB. Em cada ponto de interceptação existe apenas um instância do modelo de interceptadores. Os pontos de interceptação presentes dentro de um ORB invocam o interceptador sempre que ocorre algum tipo de evento dentro do ORB. Então o interceptador deve verificar se existe a necessidade de aplicar alguma propriedade não-funcional. A definição do ponto dentro do ORB em que uma propriedade não-funcional deve ser aplicada pelo interceptador dependerá do tipo de informação do estado interno do ORB disponível naquele ponto de interceptação.

Por exemplo, para a inclusão de tolerância a falhas através de réplicas, esta propriedade não-funcional deve ser colocada no ponto do ORB em que é possível acessar as requisições que estão sendo criadas. Em um ORB cliente baseado no padrão CORBA,

este ponto estaria entre o *Stub* e o *Núcleo do ORB*. Assim, o interceptador poderia criar uma cópia da requisição e a enviá-la para uma réplica do objeto remoto que está em algum ORB servidor. Caso houvesse uma falha no objeto remoto original, o interceptador captaria a resposta da réplica.

Em outro caso, para a inclusão de criptografia ou compressão dos dados, estas propriedades não-funcionais devem ser ativadas pelo interceptador no ponto do ORB cliente em que é possível acessar os fluxos de dados que estão sendo enviados para a rede. Então um outro interceptador deve estar posicionado no local de onde se pode acessar os fluxos de dados recebidos da rede pelo ORB servidor para a aplicação destas propriedades. Num ORB baseado no padrão CORBA, este pontos estariam entre o *Núcleo do ORB* e a *interface* da rede.

3.4 Implementação

A implementação do modelo de interceptadores dinâmicos foi feita para ser utilizada dentro do OiL [38], que é um ORB baseado em CORBA e implementado na linguagem Lua [27]. A seguir, são mostrados como os componentes do modelo de interceptadores dinâmicos foram implementados no OiL e também são descritas as vantagens da utilização da linguagem Lua.

3.4.1 Implementação dos Pontos de Interceptação e do Componente *Context*

Na Figura 3.3, é mostrado um exemplo de envio de uma requisição do cliente, passando pelos principais componentes do OiL, chegando até o servidor e, posteriormente, o envio da resposta para o cliente. Ao longo do caminho percorrido, tanto pela requisição quanto pela resposta e também dentro de alguns componentes do OiL, existem pontos de interceptação implementados na sua estrutura, nos quais podem ser inseridos interceptadores dinâmicos.

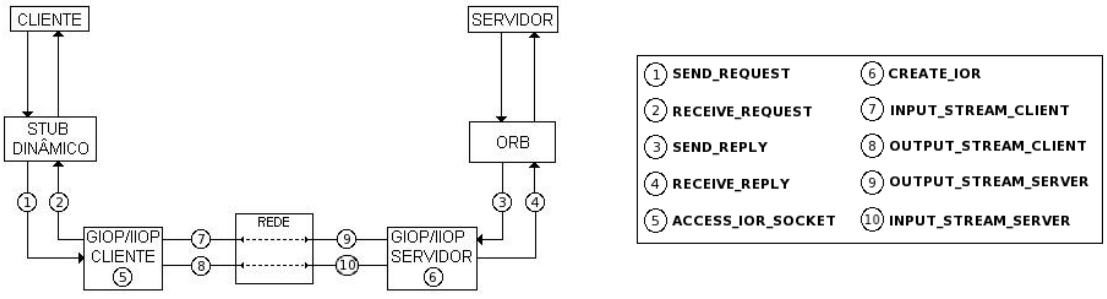

Figura 3.3: *Pontos de Interceptação dentro ORB OiL*

Como foi citado na seção anterior, a definição do ponto dentro do ORB em que uma propriedade não-funcional deve ser aplicada por um interceptador dinâmico, dependerá do tipo de informação disponível sobre o estado interno do ORB neste ponto. Os pontos de interceptação mostrados na Figura acima são descritos abaixo:

- Os pontos *Send_Request* e *Receive_Request* no lado do cliente e os pontos *Send_Reply* e *Receive_Request* no lado do servidor servem para ter acesso aos parâmetros e resultados de uma requisição.
- O ponto *Access_IOR_Socket* dentro do componente *GIOP/IIOP Cliente* do OiL serve para acessar uma IOR [48] obtida pelo OiL no lado do cliente. Este ponto permite também o acesso ao *socket* utilizado para se conectar ao ORB Servidor.
- O ponto *Create_IOR* dentro do componente *GIOP/IIOP Servidor* do OiL serve para acessar as IORs geradas pelo OiL no lado do servidor.
- Os pontos *Output_Stream_Client* e *Input_Stream_Client* no lado do cliente e os pontos *Output_Stream_Server* e *Input_Stream_Server* no lado do servidor podem ser utilizados para acessar os fluxos de dados enviados e recebidos pela rede.

Esta identificação dos pontos de interceptação por nomes simbólicos tem grande importância na implementação do modelo de interceptadores dinâmicos. Através destes nomes, o desenvolvedor poderá saber o ponto de interceptação adequado para aplicar uma propriedade não-funcional.

O componente *Context* foi implementado como uma tabela da linguagem Lua que recebe as informações disponíveis no local do ORB em que foi inserido um ponto de interceptação. Como mostrado no Capítulo 2, as tabelas de Lua são vetores que recebem valores de qualquer tipo desta linguagem. Da mesma forma, qualquer tipo de valor também pode ser utilizado para indexar estas tabelas. Com isso, o *Context* pode ser utilizado em qualquer local do OiL sem precisar sofrer modificações para armazenar algum tipo de dado específico.

3.4.2 Implementação do Componente *Monitor*

A definição em linguagem Lua da estrutura básica do componente *Monitor* é apresentada no Código 3.1. Toda vez que a função *Interception_Point()* (linha 57) é chamada a partir de algum ponto de interceptação, o *Monitor* verifica se um determinado arquivo Lua existe. Caso exista, esse arquivo deve conter o código responsável pelo monitoramento do ambiente de execução e do estado interno do ORB, o qual seria mais adequado ao ponto de interceptação em que foi chamado. Este código é então carregado dinamicamente pelo *Monitor*.

Depois de ser carregado, o código de monitoramento começa a procurar por alterações no ambiente de execução ou no estado interno do ORB. Se ocorrer alguma mudança, este código deve informar qual dos *Implementors* deve ser utilizado através de um valor inteiro transmitido através da variável *ID_Implementor* (linha 69 a 71).

Esta abordagem permite a um desenvolvedor inserir um arquivo com código de monitoramento específico em um ponto de interceptação sem a necessidade de alterar o código do *Monitor*. As únicas ações a serem feitas são a criação de um arquivo com o nome *<Monitoring_Code_ ><ID_Interception_Point ><.lua >* e a colocação desse arquivo no local definido pela variável *Monitoring_Codes_Path*. A variável *ID_Interception_Point* representa o ponto de interceptação que deve ser monitorado.

Um *Implementor* também é ativado dinamicamente sem que haja necessidade de alterar o código do *Monitor*. Para isto, basta apenas criar um arquivo que contenha o código do *Implementor* e possua o nome *<Implementor_ ><ID_Interception_Point >_ <ID_Implementor ><.lua >*. A variável *ID_Implementor* identifica o *Implementor* enquanto que a variável *ID_Interception_Point* representa o ponto de interceptação em que o *Implementor* pode ser aplicado. Em seguida, este arquivo deve ser colocado no local definido pela variável *Implementors_Path*.

Para exemplificar esta abordagem, considere o seguinte caso. Um desenvolvedor deseja monitorar os fluxos de dados que são enviados do ORB Cliente para o ORB Servidor para a aplicação de alguma propriedade não-funcional. O código de monitoramento deve ser então ativado no ponto de interceptação *Input_Stream_Client*. Desta forma, o nome do arquivo que contém esse código de monitoramento será *Monitoring_Code_Input_Stream_Client.lua*, que deve estar no diretório definido pela variável *Monitoring_Codes_Path*. Quando acontece uma determinada mudança no ambiente de execução, o código de monitoramento retorna um identificador de valor 1. Isto significa que o código da propriedade não-funcional requisitada está no arquivo *Implementor_Input_Stream_Client_1.lua* localizado num diretório definido pela variável *Implementors_Path*.

Código 3.1: *Componente Monitor*

```
1  -- FUNÇÕES DA BIBLIOTECA DA LINGUAGEM LUA UTILIZADAS PARA
2  -- CARREGAR E ATIVAR UM CÓDIGO LUA EM TEMPO DE EXECUÇÃO
3  local loadfile = loadfile
4  local pcall    = pcall
5  ------------------------------------------------------------------------
6  local string    = require "string"
7
8  module "oil.dynamicInterceptor.Monitor"
9
10 -- O MONITOR USA ESTE MÓDULO PARA GRAVAR E LER ARQUIVOS
11 -- EM UMA MEMÓRIA CACHE
```

```
12 local Cache      = require "oil.dynamicInterceptor.Cache"
13
14 -- FUNÇÃO PARA CARREGAR E ATIVAR ALGUM CÓDIGO (IMPLEMENTOR OU MONITORAMENTO)
15 -- EM TEMPO DE EXECUÇÃO
16 local function Activate_Code_Runtime (Context, Code_Path)
17
18    local Result
19
20    -- VERIFICA SE ESTE ARQUIVO JÁ ESTÁ NA CACHE
21    local Code = Cache.Search_in_Cache(Code_Path)
22
23    if Code == nil then -- CASO NÃO ESTEJA
24       Code, Error = loadfile(Code_Path) -- CARREGA O CÓDIGO DENTRO DO
25                                         -- ARQUIVO
26
27       if not Code then  -- VERIFICA A EXISTÊNCIA DO ARQUIVO
28          print("Error loading file.\n" .."Error: ".. Error .. "\n")
29          return nil -- CASO NÃO EXISTA É RETORNADO VALOR NULO
30       end
31
32       -- CASO O ARQUIVO EXISTA
33       Cache.Add_File_in_Cache(Code, Code_Path) -- ENTÃO É ARMAZENADO NA CACHE
34    end
35
36
37    Code, Result = pcall(Code) -- OBTÉM A FUNÇÃO PRINCIPAL
38
39    if not Code then
40       print("Error obtaining the function\n".."Error: ".. Result .. "\n")
41
42    else
43       Code, Result = pcall(Result, Context) -- CHAMA A FUNÇÃO PRINCIPAL
44
45       if not Code then
46             print("Error calling the function\n".."Error: ".. Result .. "\n")
47          end
48
49       end
50    end
51
52    return Result
53
54 end
55
56 -- FUNÇÃO CHAMADA PELO ORB QUANDO ATINGE UM PONTO DE INTERCEPTAÇÃO
57 function Interception_Point(Context, ID_Interception_Point)
```

```
58
59 ------------------------------------------------------------------------
60 -- VARIÁVEL COM A STRING DO NOME DO CÓDIGO DE MONITORAMENTO
61     local Monitoring_Code_Name = "Monitoring_Code_"..ID_Interception_Point..".lua"
62
63 -- NOME E LOCAL DO CÓDIGO RESPONSÁVEL PELO MONITORAMENTO DO
64 -- AMBIENTE DE EXECUÇÃO E DO ESTADO INTERNO DO ORB
65     local Monitoring_Code_Full_Name = Monitoring_Codes_Path..Monitoring_Code_Name
66
67 ------------------------------------------------------------------------
68 -- INFORMA QUAL IMPLEMENTOR DEVE SER CARREGADO
69     local ID_Implementor = -- Esta variável possui um inteiro que
70                            -- o IMPLEMENTOR a ser carregado
71     Activate_Code_Runtime(Context, Monitoring_Code_Full_Name)
72
73 -- CASO NENHUM IMPLEMENTOR TENHA SIDO ESCOLHIDO OU ARQUIVO COM O CÓDIGO DE
74 -- MONITORAMENTO NÃO EXISTA
75     if not ID_Implementor then
76         return -- O MONITOR TERMINA SUA EXECUÇÃO
77     end
78
79 -- VARIÁVEL COM A STRING DO NOME DO IMPLEMENTOR
80     local Implementor_Name = "Implementor_"..ID_Interception_Point..ID_Implementor..".lua"
81
82 -- NOME E LOCAL DO CÓDIGO DO IMPLEMENTOR RESPONSÁVEL POR IMPLEMENTAR
83 -- ALGUMA PROPRIEDADE NÃO-FUNCIONAL
84     local Implementor_Full_Name = Implementors_Path..Implementor_Name
85
86 -- CARREGA O IMPLEMENTOR
87     Activate_Code_Runtime(Context, Implementor_Full_Name)
88
89 end
```

A função *Activate_Code_Runtime()*

A função *Activate_Code_Runtime()* é utilizada pelo *Monitor* para carregar e ativar o código de monitoramento e também o código do *Implementor* em tempo de execução. Antes de tentar carregar o arquivo com o código, a função *Activate_Code_Runtime()* verifica se este arquivo já está carregado através da função *Search_in_Cache* (linha 21) do módulo auxiliar *Cache*. Este módulo serve para acessar arquivos que já foram carregados antes. Se estiver carregado, o arquivo é passado para a variável *Code*. Caso contrário, o arquivo com o código é carregado utilizando a função *loadfile* (linha 24). Esta função da biblioteca da linguagem Lua recebe como parâmetro de entrada, uma *string* contida na

variável *Code_Path* contendo o local no qual está o arquivo e o seu nome com o código que deve ser carregado em tempo de execução.

Se não houver erros dentro do código, esta função retorna o código compilado dentro de uma outra função. Como foi explicado no Capítulo 2, para ativar dinamicamente um código em que é necessário passar parâmetros e também receber resultados da sua função principal, a função *pcall()* tem que ser chamada duas vezes. Na primeira vez, *pcall()* retorna a função principal do código para a variável *Result* (linha 37). Na segunda vez, *pcall()* chama a função principal contida na variável *Result*, passando como parâmetro o *Context* que pode ser usado tanto pelo código de monitoramento quanto pelo código do *Implementor* (linha 43). Em seguida, o valor retornado pela função principal, caso haja algum, é passado também para a variável *Result*, que por sua vez é retornado para o *Monitor* (linha 52).

3.4.3 Implementação do Componente *Implementor*

O código 3.2 apresenta a estrutura básica de um *Implementor* na qual deve estar definida uma propriedade não-funcional. Como podemos observar, a sua função principal deve receber como parâmetro a tabela *Context*. Outra característica importante do *Implementor* é que seu código sempre retorna sua função principal. Desta forma, o *Implementor*, ao ser carregado dinamicamente, poderá receber a tabela *Context* cujos valores podem ser úteis na implementação desta propriedade.

Código 3.2: *Componente Implementor*

```
1  function implementacao_Propriedade_NaoFuncional
2                                          (Context)
3
4  -- Implementação de uma propriedade não-funcional
5  -- carregada dinamicamente através do
6  -- componente 'Monitor'
7
8      return implementacao_Propriedade_NaoFuncional
9
10 end
```

3.5 Comparações do Modelo de Interceptadores com outros Trabalhos

Nesta seção, são feitas comparações do modelo de interceptadores dinâmicos com o AutoGrid e também com o AutoMate. Estes trabalhos, assim como o modelo de

interceptadores, têm por objetivo ajudar as plataformas de *middleware* de grade a lidar com o dinamismo das grades computacionais.

A abordagem apresentada por estes trabalhos consiste em modificar a estrutura de um sistema de *middleware* de grade para a inclusão de novos componentes com mecanismos de adaptação dinâmica capazes de lidar com o dinamismo das grades computacionais. Além disso, estes trabalhos incluem também aspectos autonômicos, ou seja, permitem que um *middleware* de grade realize adaptações sem a necessidade de configuração por parte do usuário.

3.5.1 Comparação do Modelo de Interceptadores com o AutoGrid

O AutoGrid tem objetivos parecidos com o modelo de interceptadores dinâmicos desenvovido neste trabalho, já que os dois buscam inserir mecanismos de adaptação dinâmica ao InteGrade. Entretanto, existem diferenças na maneira em que ambos atuam para permitir que o InteGrade consiga lidar com a dinamicidade presente nas grades computacionais.

O AutoGrid pretende criar um *middleware* de grade autônomo a partir do InteGrade através da adição de novos componentes na sua infra-estrutura. Por causa disto, a arquitetura do InteGrade teve que ser alterada para poder se adequar à incorporação desses novos componentes. Já o modelo de interceptadores dinâmicos altera apenas o ORB OiL, que é um dos ORBs utilizados para comunicação entre seus componentes. Desta forma, não houve a necessidade de alteração na arquitetura do InteGrade. No entanto, ainda é necessário incorporar o modelo de interceptadores ao ORB JacORB para que o InteGrade possa usufruir melhor dos seus mecanismos de adaptação dinâmica.

As alterações feitas pelo AutoGrid na arquitetura do InteGrade possibilitaram a inclusão de quatro propriedades relacionadas a adaptação dinâmica, que são o *domínio de contexto*, a *auto-configuração*, a *auto-recuperação* e a *auto-otimização*. Através destas propriedades, o AutoGrid possui a capacidade de se adaptar às variações que ocorrem em uma grade computacional sem que haja intervenção do usuário, ou seja, de forma autônoma.

O modelo de interceptadores dinâmicos também consegue monitorar uma grade e reagir ao seu dinamismo de forma adequada. No entanto, a decisão sobre como lidar com uma determinada mudança que acontece nas grades computacionais pode depender da interferência do usuário. No Capítulo 4, serão mostrados exemplos deste tipo de situação.

3.5.2 Comparações do Modelo de Interceptadores com o AutoMate

O AutoMate é um arcabouço utilizado para o desenvolvimento de aplicações de grade a partir de componentes autônomos. Desta forma, as aplicações podem ser capazes de

configurar, gerenciar, adaptar e otimizar sua própria execução. Para que um *middleware* de grade possa executar aplicações autônomas, sua infra-estrutura deve ser estendida com os serviços definidos pelo AutoMate para gerenciamento, controle de acesso, confiabilidade e políticas de execução, entre outros.

O modelo de interceptadores dinâmicos é uma implementação que tem por objetivo incorporar mecanismos de adaptação dinâmica ao InteGrade. Como este modelo é implementado no ORB de comunicação OiL, não existe a necessidade de que as aplicações de grade sejam criadas seguindo um padrão definido por algum arcabouço. Além disso, os componentes do InteGrade não precisaram sofrer nenhuma alteração. Ainda assim, por causa do modelo de interceptadores dinâmicos, o InteGrade é capaz de utilizar técnicas de adaptação para lidar com o ambiente de execução dinâmico das grades computacionais.

A arquitetura do modelo de interceptadores dinâmicos e a arquitetura do AutoMate possuem alguns aspectos semelhantes. Por exemplo, o *Mecanismo Ciente de Contexto* serve para obter informações sobre a grade computacional e então notificar as camadas do AutoMate sobre alguma mudança. O componente *Monitor* do modelo de interceptadores também deve recolher informações da grade e ativar algum componente *Implementor* caso haja necessidade. O componente *Implementor* do modelo de interceptadores e a *Camada de Componentes* do AutoMate também são semelhantes já que ambos são responsáveis por realizar algum tipo de adaptação. No entanto, o tipo de adaptação fornecido por um componente *Implementor* é bem mais simples e restrito.

A maior diferença que pode ser notada entre as duas arquiteturas é o local de implementação. A arquitetura do AutoMate é implementada entre o *middleware* de grade e a aplicação. Já o modelo de interceptadores dinâmicos é implementado dentro do InteGrade, mais especificamente em um de seus ORBs de comunicação. O motivo do AutoMate ser definido nesta posição serve para que uma aplicação autonômica possa ser executada por qualquer *middleware* de grade baseado no padrão OGSA [43], como por exemplo o Globus [23, 18]. No caso do modelo de interceptadores, o local de sua implementação serve para possibilitar que o InteGrade tenha capacidade de adaptação dinâmica sem que haja necessidade de adicionar um novo componente. Outra diferença importante é o fato de que o AutoMate, assim como o AutoGrid, consegue lidar com o dinamismo das grades computacionais de forma autônoma.

3.6 O Modelo de Interceptadores no JacORB

A inclusão do modelo de interceptadores dinâmicos no JacORB é importante pois permitirá aos componentes do InteGrade que utilizam este ORB aproveitarem os benefícios fornecidos por este mecanismo de adaptação dinâmica. A linguagem Java, utilizada para implementar o JacORB, oferece suporte para carregamento e ativação de código

dinamicamente, assim como Lua. Portanto, o modelo de interceptadores, implementado na forma de objetos Java, também poderá ativar uma propriedade não-funcional no JacORB em tempo de execução. No entanto, existem alguns outros aspectos do JacORB que podem tornar mais difícil a implementação do modelo de interceptadores.

Por ser maior e mais complexo que o OiL, a definição dos pontos de interceptação dentro JacORB será mais complicada. Isto significa que vai ser mais difícil encontrar um local do JacORB no qual é possível inserir um ponto de interceptação para acessar informações tais como o *socket* utilizado para conexão com um outro ORB. No entanto, os interceptadores de CORBA [48] poderiam ajudar na implementação destes pontos de interceptação.

Para isso, os interceptadores de CORBA, cujo suporte é oferecido pelo JacORB, seriam utilizados como pontos de interceptação. De forma similar ao que foi feito em [39], os interceptadores de CORBA seriam utilizados para ativar dinamicamente uma implementação do modelo de interceptadores. Assim, os interceptadores de CORBA poderiam ser utilizados para ter acesso às requisições e respostas transmitidas entre um cliente e o servidor e também às referências de um objeto remoto. Entretanto, o acesso a informações tais como os fluxos de dados não são fornecidas pelos interceptadores de CORBA. Por isso, para obter estas outras informações, a inserção de outros pontos de interceptação deve ser feita diretamente no JacORB, sem o auxílio dos interceptadores de CORBA.

Outra diferença será a complexidade da implementação do modelo de interceptadores no JacORB. Como o OiL foi feito usando Lua, as informações de seu estado interno estão dentro de variáveis e tabelas que podem ser acessadas diretamente pelo modelo de interceptadores. Além disso, não existe a necessidade de se preocupar com a definição dos tipos dessas variáveis e tabelas, pois a linguagem Lua é fracamente tipada. Por isso, uma mesma variável pode ser utilizada para armazenar qualquer tipo de valor.

O JacORB, por sua vez, foi implementado em Java. Portanto, as informações de seu estado interno vão estar definidas como atributos dentro de objetos. Com isto, essas informações só poderão ser obtidas pelo modelo de interceptadores através de métodos desses objetos. Além disso, a linguagem Java é fortemente tipada, ou seja, uma variável pode armazenar somente um tipo de valor. Devido a estas diferenças entre Lua e Java, pode-se concluir que a implementação do modelo de interceptadores no JacORB será mais difícil.

3.7 O Modelo de Interceptadores no OiL baseado em Componentes

O OiL é uma plataforma de *middleware* que visa fornecer suporte a adaptação dinâmica. Por causa disso, vários estudos têm sido feitos para possibilitar que o OiL atinja esta meta. O trabalho, descrito em [41], apresenta uma versão do OiL baseada em componentes, que tem por objetivo simplificar a inclusão de diferentes formas de adaptação em sua arquitetura, como por exemplo interceptadores. Na implementação desta versão, foi adotado um modelo de componentes chamado LuaCCM [37].

Desta forma, um dos possíveis modos de implementar o modelo de interceptadores dinâmicos nesta versão do OiL seria na forma de componentes. Uma instância baseada em componentes do modelo de interceptadores poderia ser colocada no ponto de comunicação entre dois componentes do OiL. Com isso, todas as informações trocadas entre estes dois componentes seriam captadas pelo modelo de interceptadores. A partir dessas informações, o modelo de interceptadores teria condições de modificar a maneira na qual estes componentes comunicam entre si, permitindo ao OiL lidar com as variações no ambiente do execução e também no seu estado interno com relação às propriedades não-funcionais.

Outro modo de implementar o modelo de interceptadores dinâmicos na versão componentizada do OiL seria através do suporte oferecido pelo LuaCCM. Neste caso, um interceptador deve ser inserido como um objeto Lua dentro das portas de um componente. Portas são os meios pelos quais os componentes do OiL se comunicam entre si. Cada componente possui pelo menos um tipo de porta chamada faceta e também um outro tipo chamado receptáculo [37].

As facetas são responsáveis por disponibilizar os serviços oferecidos por um componente. Já os receptáculos servem para definir quais serviços um componente necessita para o seu funcionamento. Para que a comunicação entre componentes seja possível, a faceta de um componente deve estar conectada ao receptáculo de outro.

Assim, uma instância do modelo de interceptadores que estivesse presente, como um objeto Lua, dentro de uma faceta ou receptáculo de um componente, também possui acesso às informações trocadas entre esse componente e algum outro. Portanto, caso o modelo de interceptadores seja inserido no OiL utilizando o suporte provido por LuaCCM, será possível tornar o OiL capaz de manipular suas propriedades não-funcionais para se adequar as variações do ambiente do execução e também do seu estado interno.

Esta segunda abordagem para a implementação do modelo de interceptadores dinâmicos no OiL baseado em componentes é mais vantajosa. Por causa do suporte oferecido pelo LuaCCM, a inserção de interceptadores é feita causando um menor impacto na arquitetura do OiL.

A Figura 3.4 apresenta a arquitetura do OiL baseado em componentes e também os possíveis locais dentro de sua estrutura em que podem ser inseridos pontos de interceptação. Logo abaixo, temos a descrição destes locais:

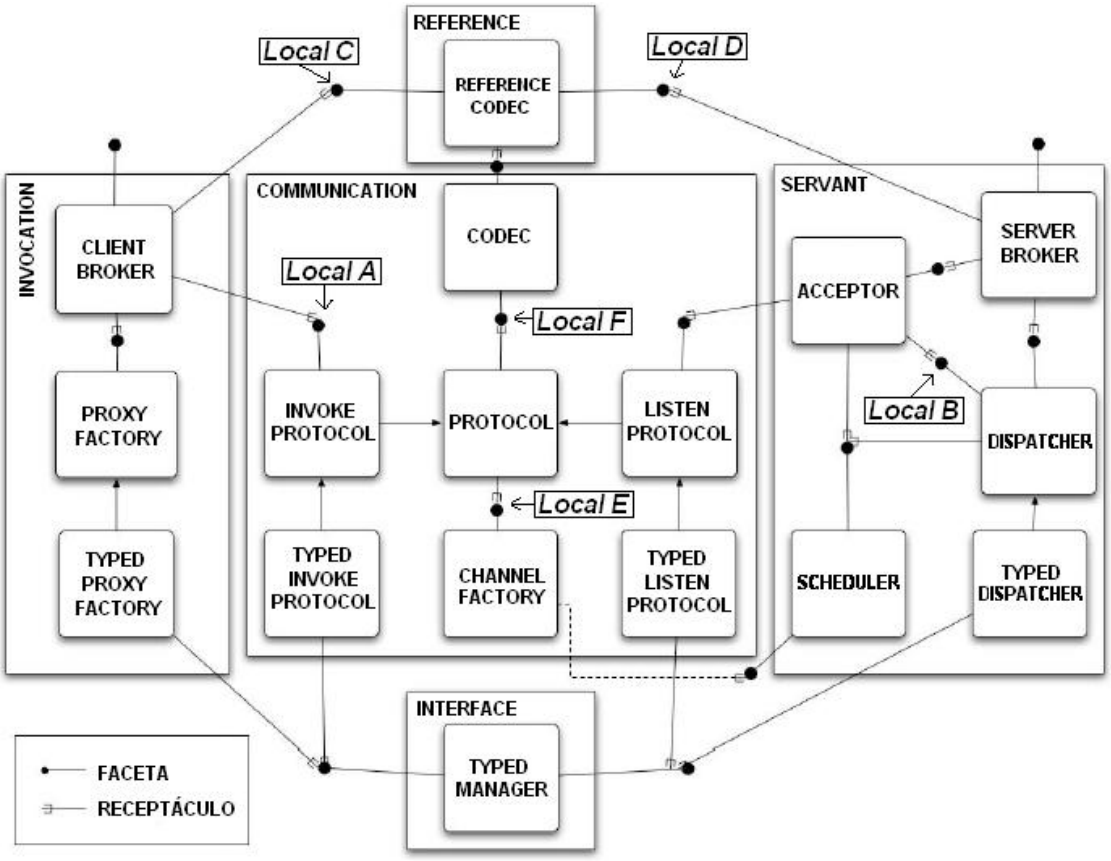

Figura 3.4: *Versão do OiL baseada em componentes*

- No local A, situado entre os componentes *ClientBroker* e *InvokeProtocol*, pode ser implementado um ponto de interceptação para acessar os parâmetros de uma requisição enviada pelo cliente para o servidor e também os resultados desta requisição retornada pelo servidor.
- No local B, situado entre os componentes *Acceptor* e *Dispatcher*, pode ser inserido um ponto de interceptação onde há necessidade de ter acesso aos parâmetros de uma requisição recebida de algum cliente e também aos resultados que serão retornados.
- No local C, situado entre os componentes *ClientBroker* e *ReferenceCodec*, um ponto de interceptação pode ser inserido para ter acesso a uma referência de objeto servidor lida pelo cliente.
- No local D, situado entre os componentes *ServerBroker* e *ReferenceCodec*, um ponto de interceptação pode ser inserido para ter acesso a uma referência de objeto gerada por um servidor.

- No local E, situado entre os componentes *Protocol* e *ChannelFactory*, pode ser inserido um ponto de interceptação em que seja necessário acessar as conexões criadas entre o cliente e o servidor.
- No local F, situado entre os componentes *Protocol* e *Codec*, pode ser implementado um ponto de interceptação onde há necessidade de ter acesso aos fluxos de dados que serão enviados e recebidos pela rede.

3.8 Considerações Finais

O modelo de interceptadores dinâmicos apresentado neste capítulo foi desenvolvido para investigar a factibilidade e os benefícios de utilizar mecanismos de adaptação dinâmica de um ORB de comunicação por um *middleware* de grade. Para isto, este modelo de interceptadores foi implementado no OiL, um dos ORBs de comunicação do InteGrade. Assim, a utilização do modelo de interceptadores possibilitou que o OiL tivesse a capacidade otimizar dinamicamente sua configuração sempre que acontecesse alguma mudança no ambiente de execução e também no seu estado interno. Então, o InteGrade, aproveitando os mecanismos de adaptação fornecidos pelo OiL, também passou a ter capacidade de adaptar sua configuração para lidar com as variações do seu ambiente de execução.

No entanto, somente os componentes do InteGrade, que utilizam o OiL, poderão se beneficiar diretamente do modelo de interceptadores. Portanto, para que o InteGrade possa usufruir completamente deste mecanismo de adaptação dinâmica, é necessária também a implementação deste modelo de interceptadores no JacORB.

Em seguida, foram feitas comparações entre o modelo de interceptadores dinâmicos com outros dois trabalhos, o AutoMate e o AutoGrid, que também buscam adicionar mecanismos de adaptação dinâmica em plataformas de *middleware* de grade. Dando seqüência, discutimos alguns aspectos da implementação do modelo de interceptadores no JacORB e também na versão componentizada do OiL.

Um aspecto importante do modelo de interceptadores consiste na capacidade observar o ambiente de execução e o estado interno de um ORB através do *Monitor*. Na implementação deste componente no OiL, o código de monitoramento, responsável por detectar alterações no ambiente de execução do InteGrade e no estado interno deste ORB, está contido num arquivo que é carregado e ativado pelo *Monitor* em tempo de execução. A vantagem no uso desta abordagem é que um desenvolvedor pode inserir um arquivo com código de monitoramento específico em um ponto de interceptação sem a necessidade de alterar o código do *Monitor*.

Para permitir uma maior segurança no uso desta abordagem, algumas medidas podem ser tomadas. Por exemplo, o acesso ao arquivo com o código de monitoramento pelo componente *Monitor* só poderia ser feito através de autenticação. Outra medida seria a

encriptação deste arquivo. Então o *Monitor* ficaria responsável por decodificar este arquivo antes de ser feito o acesso. Uma outra medida de segurança seria colocar este arquivo num local em que somente o *Monitor* teria permissão de acesso. No entanto, esta última medida só pode ser feita pelo administrador da rede.

Estes recursos de segurança poderiam ser aplicados também no arquivo que contém o código do *Implementor*, já que este tipo de arquivo também é carregado e ativado dinamicamente pelo *Monitor*. No próximo capítulo, são apresentados exemplos de aplicação do modelo de interceptadores, demonstrando os benefícios do seu uso.

CAPÍTULO 4

Aplicações do Modelo de Interceptadores Dinâmicos

Este capítulo descreve duas propriedades não-funcionais inseridas no ORB OiL através do uso do modelo de interceptadores dinâmicos. A aplicação destas propriedades tem por objetivo mostrar as vantagens no uso deste modelo de interceptadores para adicionar mecanismos que permitam ao InteGrade lidar com a alta dinamicidade presente nas grades computacionais.

4.1 Travessia de *Firewall* e NAT

Os recursos e serviços disponibilizados pelas grades computacionais geralmente estão distribuídos por múltiplos domínios administrativos, presentes em várias localidades ao redor do planeta. Devido a esta alta dispersão, diferentes instituições acadêmicas ou corporativas são responsáveis por esses domínios que constituem as grades.

Um domínio administrativo é formado por um aglomerado de máquinas sujeitas a um determinado conjunto de políticas e restrições estabelecidas por alguma instituição local. Por exemplo, as máquinas de um laboratório em um instituto de pesquisa sob controle de um administrador de rede constituem um domínio administrativo.

As políticas e restrições definidas nestes domínios administrativos têm por objetivo prover maior controle e segurança sobre seus recursos e serviços. Exemplo disto são as *firewalls* [56], utilizadas para gerenciar o fluxo de entrada e saída de informações para dentro e fora de um domínio administrativo, com intuito de proteger a rede contra ataques externos.

Entretanto o uso de *firewalls* limita a conectividade dos computadores em uma rede. Esta limitação impede que os sistemas de *middleware* de grade sejam capazes de agregar recursos de diferentes domínios administrativos. A única exceção são as plataformas de *middleware* de grade baseadas em *Web Services* [54] tais como o *Globus Toolkit 4* [26]. No caso do InteGrade, que utiliza ORBs baseados em CORBA para comunicação, não é possível executar aplicações sobre recursos que estejam distribuídos

em redes distintas pertencentes a diferentes instituições. Isto acontece porque o uso de *firewalls* entra em conflito com duas características fornecidos por CORBA que são importantes para o InteGrade: transparência de comunicação e modelo de comunicação baseada em *peers* [46].

A transparência de comunicação permite que um cliente não tenha necessidade de conhecer a exata localização de um objeto para invocá-lo. Portanto o objeto pode ser realocado para um novo servidor sem alterar qualquer informação no lado do cliente. O InteGrade, por ser um *middleware* de grade oportunista, necessita fornecer transparência de comunicação porque está sempre com seus componentes mudando de localidade à medida que as tarefas se deslocam dentro da grade computacional.

No entanto, este benefício introduz um problema quando um *firewall* é utilizado para proteger um servidor, visto que *firewalls* geralmente controlam o tráfego de entrada de dados na rede através de um conjunto de regras que definem quais endereços e portas podem ser acessados. Para permitir que um objeto mude de posição e ainda seja possível acessá-lo, uma nova configuração deve ser feita. Esta reconfiguração é necessária porque toda a vez que o objeto muda de servidor, o seu endereço e porta também serão alterados.

O modelo de comunicação baseado em *peers* também causa problemas na comunicação através de *firewalls* pelo fato de que qualquer computador na rede pode ser tanto um cliente quanto um servidor. Assim, o número de servidores poderia ser bastante elevado, o que tornaria difícil o gerenciamento e configuração do *firewall*.

Outro problema relacionado ao uso de comunicação *peer-to-peer* acontece quando uma rede utiliza NAT [56]. Este serviço cria um espaço de endereços IP privados dentro da rede interna e proíbe estes endereços de serem acessados pela rede externa. Quando um computador desta rede interna tem que enviar mensagens para a rede externa, o *servidor NAT* mapeia seu endereço local para algum endereço público pertencente ao domínio administrativo da instituição [46].

No InteGrade, este problema ocorre, por exemplo, quando uma aplicação paralela é executada por um conjunto de nós pertencentes a uma grade computacional. Estes nós, para que possam comunicar entre si, precisam conhecer a localização um do outro e caso um destes nós esteja dentro de uma rede com NAT, o endereço interno deste nó não terá significado, já que não é um endereço público e por isso não pode ser acessado a partir de uma rede externa.

Para lidar com estes problemas é descrita a seguir uma abordagem, implementada através do modelo de interceptadores dinâmicos, que possibilita aos componentes do InteGrade a capacidade de se comunicarem mesmo que estejam em redes distintas protegidas por *firewall* e NAT.

4.1.1 Abordagem da OMG para Travessia de *Firewall* e NAT

A abordagem adotada neste trabalho foi baseada na especificação da OMG para travessia de *firewall* e NAT em CORBA. Seu uso se aplica nos casos em que o objeto servidor não pode receber conexões da rede externa e também quando um cliente ORB não consegue se conectar a uma rede protegida por *firewall* e/ou NAT. A sua elaboração foi feita com base nos seguintes requisitos [11]:

- Não deve prejudicar o gerenciamento e configuração da *firewall*.
- Deve ser fácil de configurar e também transparente ao desenvolvedor de aplicações.
- Deve causar o mínimo de impacto no desempenho das aplicações.

O uso desta abordagem exige que uma porta seja aberta no *firewall*, a qual é usada para conduzir todo o tráfego do protocolo IIOP [48] que atravessa os limites da rede. Dentro da rede protegida pela *firewall*, todo o tráfego de entrada é redirecionado para um *proxy* de aplicação [11], que escuta em uma porta aberta na *firewall* e é responsável por repassar estas mensagens para o nó da rede desejado. Do mesmo modo, todo o tráfego de saída do protocolo IIOP é antes roteado para o *proxy* de aplicação, que usará a porta aberta para enviar mensagens ao seu destino final. A Figura 4.1 mostra o esquema de funcionamento desta abordagem.

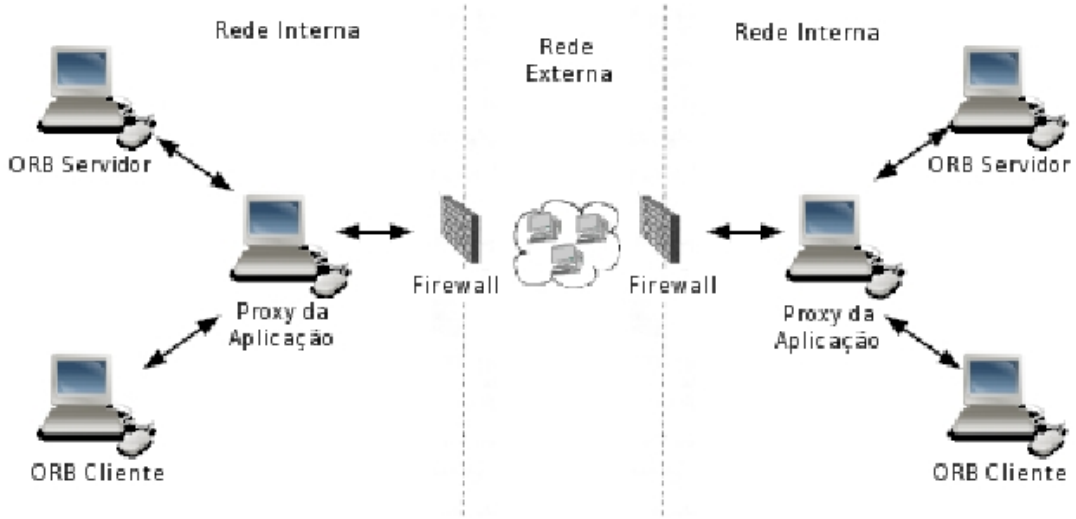

Figura 4.1: *Abordagem da OMG para Travessia de Firewall e NAT [11]*

Para que um objeto servidor possa estar acessível a partir da rede externa, a existência do *proxy* de aplicação deve ser divulgada de alguma forma. Isto é feito através de um componente rotulado que é adicionado ao perfil IIOP da IOR [48] do objeto e que contém referências a todos os nós intermediários que estejam entre a rede externa, por exemplo a Internet, e o próprio objeto. Desta forma, no caso mais simples e comum,

existiria apenas uma referência a um *proxy* da aplicação e o ORB. Quando o cliente recebe esta IOR, a informação sobre o *proxy* deve ser identificada e então deve ser criada uma mensagem do tipo *GIOP NegotiateSession* [46] mencionando todos os elementos entre o cliente e o objeto servidor. Entretanto, o cliente não deve ser mencionado nesta mensagem. Esta mensagem é então enviada seqüencialmente para todos os intermediários citados no perfil IIOP, desde o primeiro até o último antes do objeto servidor. Se a mensagem chegar até este último elemento, uma resposta será enviada para o cliente ORB anunciando que pode enviar mensagens ao objeto servidor.

A informação sobre a existência de elementos intermediários presentes no componente rotulado que está inserido dentro do perfil IIOP da IOR deve ser passada através de um arquivo de configuração fornecido pelo usuário. Então o componente rotulado é criado através deste arquivo.

Esta solução tem a vantagem de fornecer interoperabilidade entre diferentes ORBs que seguem o padrão CORBA. Esta interoperabilidade é possível pois a abordagem para travessia de *firewall/NAT* da OMG utiliza IOR, que é o formato adotado por todo ORB baseado em CORBA para especificar referências de objetos, como meio para que um objeto servidor informe aos seus clientes sobre a necessidade do uso de um *proxy* de aplicação para poder acessá-lo.

Entretanto, uma de suas desvantagens é a necessidade de uma configuração na *firewall*. Esta configuração consiste em definir uma porta que esteja aberta para receber e/ou enviar informações pela rede externa. Outra desvantagem é que a comunicação de computadores ou outros tipos de dispositivos protegidos por uma rede NAT com a rede externa só é possível se o *proxy* de aplicação estiver colocado num endereço válido tanto para a rede NAT quanto para a rede externa.

4.1.2 Implementação da Abordagem da OMG através de Interceptadores Dinâmicos

Caso a travessia de *firewall* e NAT seja implementada diretamente seguindo a definição descrita na abordagem da OMG, um cliente ORB teria que sofrer diversas modificações para ser capaz de saber quando se conectar a um ou mais *proxies* de aplicação e, assim, se comunicar com objetos servidores que estão dentro de uma rede protegida. Isto inclui também o caso em que o ORB cliente também está atrás de uma rede protegida por *firewall* e NAT. Um servidor ORB, protegido em uma rede que possui *firewall* e NAT, também teria que ser alterado para conseguir avisar que o seu acesso externo deve ser feito através de um *proxy*.

No entanto, a utilização de interceptadores dinâmicos permite que a implementação desta abordagem da OMG seja feita num ORB, tanto no lado do cliente quanto no

lado do servidor, com pequenas alterações no código e sem causar impacto na arquitetura. Estas alterações basicamente consistem na inclusão de simples chamadas para o componente *Monitor* em diferentes pontos do ORB, como descrito no Capítulo 3. Além disso, o mecanismo de travessia de *firewall* e NAT é ativado em tempo de execução apenas quando houver necessidade.

A Figura 4.2 apresenta os dois pontos dentro do ORB OiL em que foi implementada a abordagem para travessia de *firewall* e NAT utilizando o modelo de interceptadores dinâmicos. Dentro do ponto de interceptação *Access_IOR_Socket*, foi inserido um *Monitor*, que através de um código de monitoramento carregado em tempo de execução, verifica se existe a necessidade do uso de um ou mais *proxys* para acessar um servidor ORB dentro de uma rede protegida. Esta verificação é feita observando se existe um componente rotulado no perfil IIOP da IOR ou um arquivo de configuração XML no lado do cliente. Tanto este perfil IIOP quanto este arquivo devem possuir informações sobre estes *proxys*.

Caso exista a necessidade da utilização de um ou mais *proxys*, o *Monitor* carrega o *Implementor*, cuja descrição está no Capítulo 3, que redireciona a conexão para o primeiro dos *proxys* intermediários que existem entre o cliente e o servidor. Em seguida, o *Implementor* envia uma mensagem do tipo *NegotiateSession* para estes *proxys*. Somente depois de uma resposta com sucesso, mensagens de requisição normais podem ser enviadas. O motivo do *Monitor* ter sido inserido no ponto de interceptação *Access_IOR_Socket* foi porque este é o local do OiL em que um código de monitoramento pode verificar se existe um componente rotulado no perfil IIOP da IOR.

A verificação da existência de um arquivo de configuração XML no lado do cliente serve para o caso em que o ORB servidor não tenha como fornecer a informação sobre a necessidade da utilização de um ou mais *proxies* através de IOR. Então esta informação é transmitida de alguma outra forma. Em seguida, o arquivo de configuração recebe esta informação, que é lida pelo código de monitoramento naquele ponto. Como este ponto de interceptação é também um local do OiL possível de se redirecionar uma conexão, a verificação da existência de um arquivo de configuração é feita neste ponto.

No ponto de interceptação *Create_IOR*, onde ocorre o processo de criação da IOR, foi inserido um outro *Monitor*, que, utilizando também um código de monitoramento carregado dinamicamente, verifica se existe um arquivo de configuração em XML. Este arquivo contém informações sobre a existência de um ou mais nós intermediários que devem ser utilizados para que um ORB servidor possa ser acessado pela rede externa. Caso o arquivo XML exista, o *Monitor* então carrega um *Implementor* em tempo de execução. Este *Implementor* cria um componente rotulado a partir deste arquivo de configuração. Em seguida, o *Implementor* adiciona este componente dentro do perfil IIOP da IOR. Este outro *Monitor* foi inserido no ponto de interceptação *Create_IOR* porque neste local é possível para o *Implementor* inserir um componente rotulado no perfil IIOP da IOR.

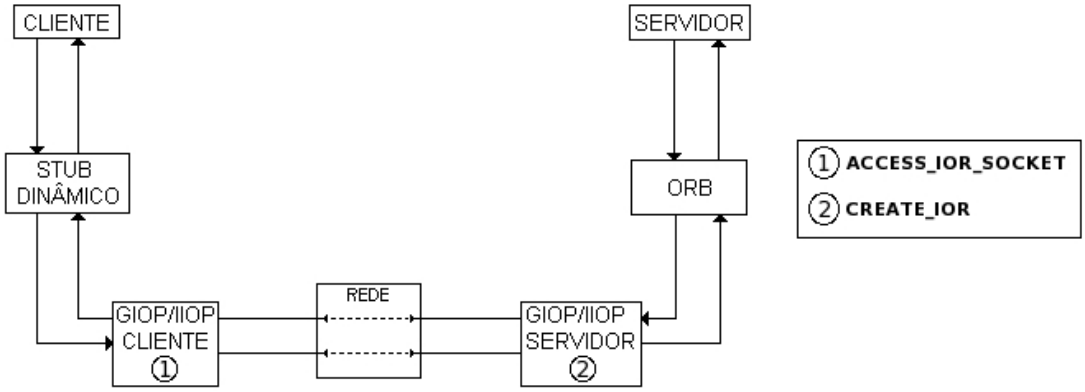

Figura 4.2: *Os pontos de interceptação utilizados para implementação de travessia de Firewall e NAT*

4.1.3 Exemplos de dois Cenários de Uso da Travessia de *Firewall* e NAT utilizando Interceptadores

A Figura 4.3 mostra o primeiro cenário de uso da travessia de *firewall* e NAT no InteGrade através do uso do modelo de interceptadores dinâmicos. Neste exemplo, um LRM está em uma rede A comum, que não possui *firewall* e NAT. Este LRM então tenta se conectar com o *Gerenciador de Recursos do Aglomerado*, localizado em uma rede B protegida por um *firewall* e que possui NAT.

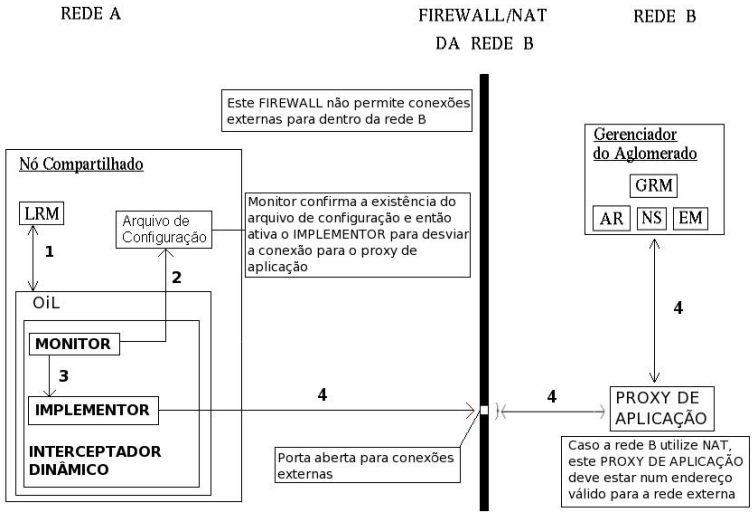

Figura 4.3: *Primeiro Cenário de Uso da Travessia de Firewall e NAT utilizando Interceptadores*

Os passos para o LRM se conectar com o *de Recursos do Aglomerado* através de um proxy, neste primeiro cenário, estão descritos abaixo:

1. O LRM utiliza o ORB de comunicação OiL para tentar se conectar com o *Gerenciador de Recursos do Aglomerado*.

2. O componente *Monitor* verifica a existência de um arquivo de configuração que contém o endereço do *proxy* de aplicação através de um código de monitoramento carregado em tempo de execução.

3. Depois de confirmar a existência do arquivo de configuração, o *Monitor* ativa o *Implementor* em tempo de execução.

4. Por sua vez, o *Implementor* redireciona a conexão para o proxy de aplicação.

Como mostrado na Figura, a porta de escuta utilizada pelo *proxy* deve estar habilitada pelo *firewall* para receber conexões externas. Somente assim o LRM conseguirá se comunicar com este *proxy* de aplicação. Para que este exemplo funcione com NAT, é necessário que o *proxy* de aplicação esteja em uma máquina com endereço válido para a rede externa.

Diante do que foi apresentado, observamos que o componente *Monitor* precisa de um arquivo de configuração para ativar o *Implementor*, que por sua vez redireciona a conexão, feita por um LRM, para o *proxy* de aplicação. Assim, quando o LRM precisa utilizar um *proxy* para se comunicar com o *Gerenciador de Recursos do Aglomerado* do InteGrade, o usuário deve inserir um arquivo de configuração.

A Figura 4.4 apresenta um segundo cenário de uso que demonstra uma outra forma para decidir se o *Implementor* é necessário para realizar o desvio de conexão para um *proxy*. Neste cenário, não existe necessidade de intervenção do usuário. Desta forma, o componente *Monitor*, ao invés de utilizar um arquivo de configuração, ativa o *Implementor* quando ocorrer um erro na tentativa de conexão.

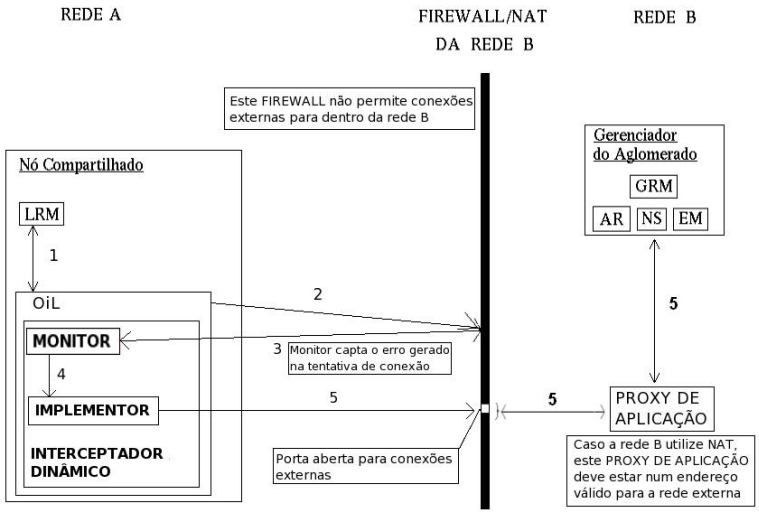

Figura 4.4: *Segundo Cenário de Uso da Travessia de Firewall e NAT utilizando Interceptadores*

Os passos para o LRM se conectar com o *Gerenciador de Recursos do Aglomerado* através de um proxy, neste segundo cenário, estão descritos abaixo:

1. O LRM utiliza o ORB de comunicação OiL para se conectar com o *Gerenciador de Recursos do Aglomerado*.
2. O ORB OiL tentar se conectar diretamente com o *Gerenciador de Recursos do Aglomerado*.
3. Como o *firewall* da rede B bloqueia qualquer tentativa externa de conexão, ocorre um erro. O componente *Monitor* capta este erro antes que ele seja passado para o OiL.
4. Em seguida, o *Monitor* ativa o *Implementor* em tempo de execução.
5. Por sua vez, o *Implementor* redireciona a conexão para o proxy de aplicação.

Com a descrição deste segundo cenário, vimos que o modelo de interceptadores dinâmicos também é capaz de ativar uma propriedade não-funcional, como a travessia de *firewall* e NAT, de forma autonômica.

4.1.4 Avaliação do Uso de Interceptadores para Travessia de *Firewall* e NAT

Descrição do Ambiente Experimental

O ambiente utilizado para a realização dos testes é o mesmo descrito no primeiro cenário apresentado na subseção anterior. Desta forma, o *Gerenciador de Recursos do Aglomerado* do InteGrade e o *proxy* de aplicação foram inseridos em duas máquinas que estão numa rede protegida por *firewall*. Entretanto, esta rede não utiliza NAT. Já o LRM foi colocado na terceira máquina, que faz parte de uma rede comum, sem *firewall*. A Tabela 4.1 detalha as características de hardware e software das máquinas utilizadas para os experimentos.

Tabela 4.1: *Características de Hardware e Software do Ambiente Experimental*

Processador	Pentium IV 3.2 GHz HT
Memória RAM	1024 MB
Disco Rígido	160 GB
Interface de Rede	Realtek 8169
Sistema Operacional	Linux Slackware Versão 11.0.0 i386 Kernel 2.6.18
InteGrade	Versão 0.3-RC1
OiL	Versão 0.3.1-Alpha
Linguagem Lua	Versão 5.0.2

A topologia da rede para a realização dos testes é mostrada na Figura 4.5. Como podemos observar, as máquinas da rede A estão conectadas com as outras máquinas da

rede B através de um *switch*. No entanto, entre este *switch* e as máquinas da rede B, existe um *firewall*.

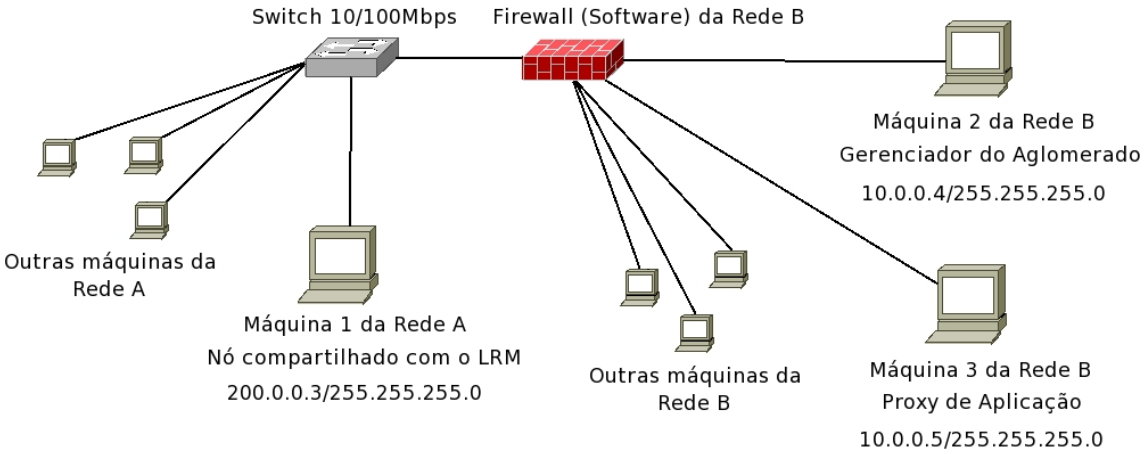

Figura 4.5: *Topologia da rede utilizada nos testes*

Medição do *Overhead*

Os resultados da medição do *overhead* foram obtidos através da análise de duas operações habituais do LRM, que são a sua inicialização e a requisição de execução de uma aplicação simples. Em cada uma destas operações, foram realizados três testes cuja descrição está definida abaixo.

- Teste I: Execução de uma operação no LRM sem a presença do modelo de interceptadores dinâmicos no InteGrade.
- Teste II: Execução de uma operação no LRM apenas com o modelo de interceptadores dinâmicos, ou seja, sem nenhuma propriedade não-funcional implementada.
- Teste III: Execução de uma operação no LRM com o modelo de interceptadores dinâmicos e também com a implementação da travessia de *firewall* e NAT através de um *proxy* de aplicação.

Os resultados dos testes para a operação de inicialização do LRM são mostrados na Tabela 4.2. Para cada teste foram feitas 50 chamadas. Nesta Tabela são apresentados o tempo máximo, o tempo de mínimo e o tempo médio para a inicialização do LRM. A medição foi feita em milisegundos (ms).

Tabela 4.2: *Medição do Overhead Para a Inicialização do LRM*

Teste Realizado	Tempo Máximo	Tempo Médio	Tempo Mínimo
Teste I	820 ms	810 ms	800 ms
Teste II	880 ms	850 ms	830 ms
Teste III	910 ms	860 ms	840 ms

Por sua vez, os resultados dos testes para a operação de requisição de execução de uma aplicação simples são apresentados na Tabela 4.3. Para cada um destes testes também foram feitas 50 chamadas e, da mesma forma, são apresentados o tempo máximo, o tempo de mínimo e o tempo médio para requisitar a execução de uma aplicação simples ao LRM. Logo abaixo é mostrada a descrição dos testes realizados. A medição foi feita em milisegundos (ms).

Tabela 4.3: *Medição do Overhead Para a Requisição de Execução de uma Aplicação Simples*

Teste Realizado	Tempo Máximo	Tempo Médio	Tempo Mínimo
Teste I	20 ms	10 ms	0 ms
Teste II	20 ms	10 ms	0 ms
Teste III	50 ms	20 ms	0 ms

Diante do que foi mostrado nas Tabelas acima, percebemos que houve um aumento no tempo para inicializar um LRM quando o modelo de interceptadores dinâmicos está inserido. Com a travessia de *firewall* e NAT, o tempo de inicialização sobe mais um pouco.

A Tabela 4.2 mostra que este aumento causado pelo modelo de interceptadores e também pela implementação da travessia de *firewall* e NAT foi pequeno. Por exemplo, no terceiro teste, o acréscimo no tempo médio de inicialização do LRM foi de apenas 6% em relação ao primeiro teste.

Na Tabela 4.3, apenas a presença do modelo de interceptadores não causou um aumento no tempo para a requisição de aplicação simples como mostrado no segundo teste. No entanto, houve um grande aumento de tempo quando o *proxy* de aplicação é utilizado para a travessia de *firewall* e NAT. Apesar deste aumento ter sido alto, o tempo para a execução desta operação se manteve relativamente pequeno. Desta forma, observamos que estes aumentos causados pelo modelo de interceptadores e a implementação da travessia de *firewall* e NAT causam um pequeno impacto no desempenho do InteGrade.

Entretanto, caso o modelo de interceptadores e a implementação da travessia de *firewall* e NAT sejam utilizados num ambiente de larga escala, com vários LRMs tentando se comunicar com um *Gerenciador de Aglomerado* localizado em um rede protegida,

o desempenho da rede possivelmente será prejudicado. Esta situação pode acontecer se apenas um *proxy* de aplicação for responsável pela transmissão das informações. Desta forma, este *proxy* vai causar um ¨ gargalo ¨ na rede. Para evitar que isto ocorra, outros *proxies* de aplicação devem ser disponibilizados.

4.1.5 Abordagem *Proxy* TCP para Travessia de *Firewall* e NAT

Uma outra abordagem que poderia ser aplicada através do modelo de interceptadores é a do tipo *Proxy* TCP [11]. Esta abordagem é direcionada para os casos em que um objeto servidor não pode receber conexões vindas da rede externa. No entanto, este objeto pode se conectar com os elementos da rede externa utilizando conexões TCP.

Para utilizar esta abordagem, é necessário definir um *proxy* TCP na rede externa. Então os objetos servidores, que estão em uma rede protegida, poderão se conectar com a rede externa através deste *proxy*. Esta conexão deve utilizar o protocolo TCP e pode ser mantida aberta enquanto houver necessidade de um objeto da rede protegida ser acessado por elementos da rede externa.

Depois de ser criado, um objeto servidor, para ser acessado pela rede externa, deve enviar uma mensagem de registro para o *proxy* TCP, recebendo como resposta uma IOR. Através desta IOR, um cliente na rede externa conseguirá acessar este objeto da rede protegida utilizando o *proxy* TCP. Então, todas as informações trocadas entre um cliente externo e objetos da rede protegida terão que passar por este *proxy* TCP.

A Figura 4.6 mostra um cenário de uso desta abordagem. As setas indicadas pelo número 1 mostram as conexões abertas pelos objetos da rede protegida por *firewall*, que estão dentro dos servidores ORB, durante sua inicialização. Já as setas indicadas pelo número 2 representam a conexão feita por um cliente ORB da rede externa depois que recebeu a IOR do objeto da rede protegida por *firewall*.

Existem dois tipos de conexões que são feitas pelo objeto servidor para o *proxy* TCP. A primeira conexão serve para este objeto se registrar neste *proxy*. A segunda conexão, feita logo depois da primeira, serve como meio para transmitir tanto as requisições dos clientes externos quanto as respostas que serão retornadas, sendo que estas informações sempre passarão pelo *proxy* TCP.

A maior vantagem desta abordagem é o fato de que não existe a necessidade de configuração da *firewall* e NAT. Outra vantagem é que apenas o servidor ORB deve ser modificado para a aplicação desta abordagem. Entretanto, sua desvantagem é a falta de escalabilidade do *proxy*, já que deve haver uma conexão para cada objeto servidor.

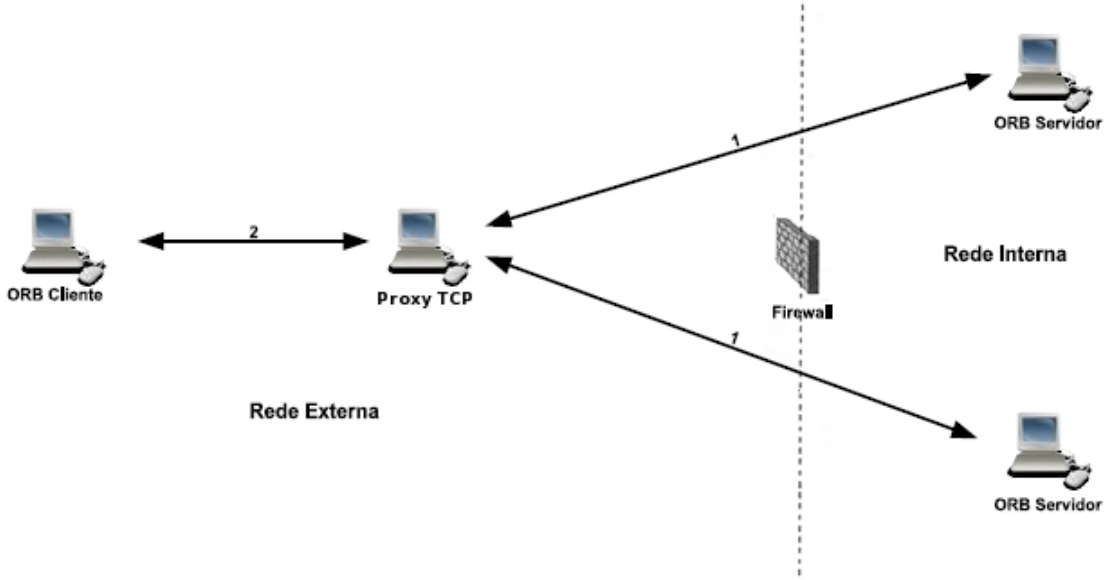

Figura 4.6: *Abordagem Proxy TCP para Travessia de Firewall e NAT [11]*

4.1.6 O Modelo de Interceptadores para Aplicação da Abordagem *Proxy* TCP

Caso o modelo de interceptadores seja utilizado para implementar a abordagem *Proxy* TCP no OiL, as alterações feitas no código deste ORB seriam mínimas, assim como foi na implementação da abordagem OMG para travessia de *firewall* e NAT. Entretanto, haveria a necessidade da definição de um novo ponto de interceptação dentro do OiL.

A Figura 4.7 mostra onde este ponto de interceptação deveria estar localizado dentro do OiL. Neste ponto seria possível ter acesso aos objetos que são criados dentro do servidor ORB. Então, dentro deste ponto de interceptação, poderia ser inserido um *Monitor* que iria verificar se haveria necessidade de se conectar com um *proxy* TCP para que um objeto recém-criado pudesse ser acessado pela rede externa. Esta verificação poderia ser feita observando se existe um arquivo de configuração XML no lado do servidor, da mesma forma que foi feito na implementação da abordagem da OMG.

Se houvesse necessidade, então o *Monitor* carregaria um *Implementor*, que ficaria responsável por se conectar ao *proxy* TCP para registrar um objeto servidor recém-criado. Em seguida, o *Implementor* criaria uma segunda conexão com o *proxy* TCP para permitir que clientes externos se comuniquem com o objeto servidor.

4.2 Freqüência de Atualização de Informações da Grade

Como foi explicado no Capítulo 2, o InteGrade é um *middleware* de grade oportunista e, por isso, precisa estar informado sobre a disponibilidade de recursos nas

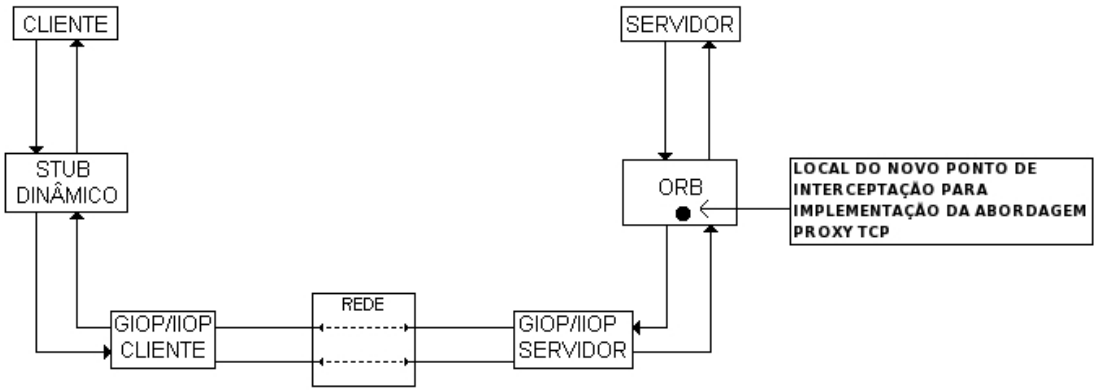

Figura 4.7: *Ponto de Interceptação para a implementação da abordagem Proxy TCP*

máquinas que compõem a grade computacional. Entretanto, algumas dessas informações, tais como quantidade de memória livre variam ao longo do tempo.

Desta forma, os LRMs presentes nas máquinas que compartilham seus recursos com a grade, precisam informar ao GRM sobre a disponibilidade de recursos com uma certa periodicidade ou, então, se houver uma mudança significativa repentina.

A freqüência com que o LRM atualiza o GRM sobre a disponibilidade de recursos em uma grade computacional possui suas vantagens e desvantagens. Caso a freqüência de atualização de recursos esteja muito alta, haverá um queda de desempenho do sistema mas as informações sobre disponibilidade de recursos serão bastante precisas. Por outro lado, se a freqüência de atualização for pequena, o desempenho será melhor, embora as informações possam ficar defasadas em relação aos recursos que a grade realmente oferece no momento.

Além disso, para determinar se a freqüência de atualização sobre a disponibilidade de recursos deve ser alta ou baixa, existem outros fatores. Por exemplo, se um LRM estiver rodando em um dispositivo portátil, tal como um PDA ou um *laptop*, um fator que poderia levar a alterar a freqüência de atualização de recursos seria a bateria. Em um determinado momento no qual a bateria está com pouca energia, seria conveniente que houvesse uma diminuição na freqüência de atualização sobre os recursos disponíveis oferecidos pelo dispositivo. Assim, o uso da *interface* de rede poderia ser menor, o que contribuiria para uma maior economia de energia da bateria.

4.2.1 Uso de Interceptadores Dinâmicos para Alterar a Freqüência de Atualização

O modelo de interceptadores dinâmicos pode ser utilizado para alterar a freqüência de atualização de forma transparente e dinâmica. O nível de energia da bateria seria uma

boa opção para decidir quando a freqüência deve ser alterada. No entanto, como a versão do LRM utilizada não é capaz de ser executada em um dispositivo portátil, o critério adotado nesta implementação foi a porcentagem de utilização da CPU.

A Figura 4.8 apresenta o ponto dentro do ORB OiL em que foi inserido um interceptador dinâmico para permitir alterar a freqüência de atualização de recursos. No ponto de interceptação *Send_Request*, onde as requisições são criadas, foi inserido um *Monitor* responsável por verificar o nível de utilização da CPU. Caso a porcentagem de uso esteja alta, um *Implementor*, carregado pelo *Monitor*, causa um atraso no envio das requisições destinadas ao GRM que servem para informar sobre a disponibilidade de recursos da máquina. Desta forma, a freqüência de atualização diminui, o que resulta em uma melhora do desempenho do sistema.

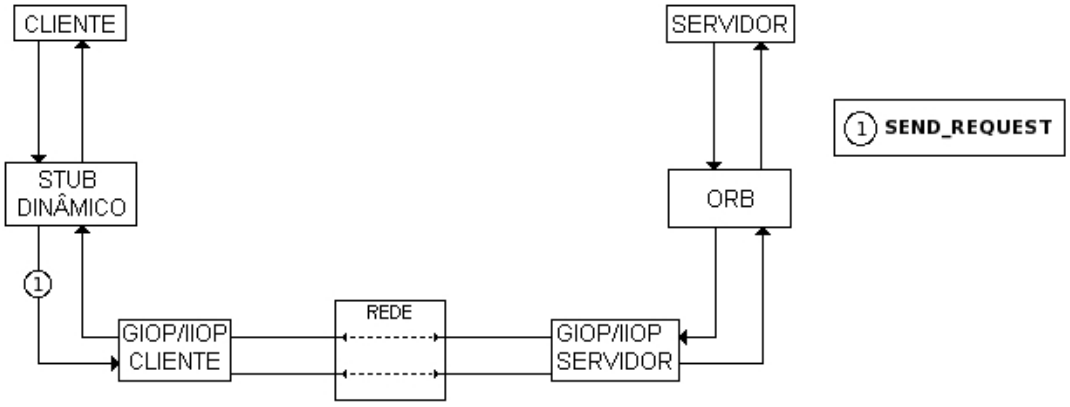

Figura 4.8: *Ponto de interceptação utilizado para diminuir a freqüência de atualização quando necessário*

4.3 Considerações Finais

Neste capítulo foram demonstradas as vantagens do uso do modelo de interceptadores dinâmicos para a introdução de duas propriedades não-funcionais no InteGrade em tempo de execução e também como os interceptadores foram inseridos dentro do ORB OiL. A primeira vantagem de sua utilização é a alteração mínima necessária na implementação do ORB para a sua inclusão. A única modificação necessária é a adição de uma simples chamada ao *Monitor* dentro do código do ORB, como discutido no Capítulo 3.

A segunda vantagem está no fato de que as propriedades não-funcionais são ativadas somente quando ocorre uma determinada mudança no ambiente de execução. Quando se executa um ORB contendo interceptadores dinâmicos, apenas o *Monitor* é inicializado. Então o *Monitor* começa a verificar a ocorrência de mudanças no ambiente de execução para decidir se uma propriedade não-funcional, definida através de um *Implementor*, deve ser ativada ou não.

Como prova disso, vimos que a abordagem da OMG para travessia de *firewall* e NAT pôde ser implementada sem que houvesse necessidade de modificar o ORB OiL. Outra vantagem é a possibilidade de retirar esta propriedade não-funcional, caso não seja mais necessária, sem modificar código e evitando a recompilação do OiL. Isto se deve ao fato de que as propriedades não-funcionais são ativadas em tempo de execução pelo modelo de interceptadores dinâmicos. O mesmo argumento também é válido para implementação do mecanismo de alterar a freqüência de atualização de informações do LRM para o GRM.

Por isso, diante do que foi apresentado, podemos dizer que, devido ao modelo de interceptadores dinâmicos, o *Gerenciador de Recursos do Aglomerado* do InteGrade pode ser acessado pelos LRMs mesmo estando dentro de uma rede protegida por *firewall* e NAT.

Entretanto, caso estes LRMs estejam dentro de outras redes, também protegidas por *firewall* e NAT, o *Gerenciador de Recursos do Aglomerado* não será capaz de se conectar com os LRMs. Por isso, é importante a implementação do modelo de interceptadores no JacORB que é o ORB utilizado pelos componentes do *Gerenciador de Recursos*, que foram implementados em Java, tais como o GRM.

Assim, os componentes do *Gerenciador de Recursos* poderão utilizar o JacORB para se comunicar com LRMs que estejam dentro de uma rede com *firewall* e NAT. Para isso, o JacORB também deve utilizar a abordagem da OMG para travessia de *firewall* e NAT que será ativada pelo modelo de interceptadores.

Outro mecanismo que o InteGrade passa a possuir também, devido ao modelo de interceptadores, é a capacidade de diminuir a freqüência na qual o LRM envia informações para o GRM. Em situações em que o LRM esteja sendo executado num dispositivo portátil, este mecanismo pode ser utilizado para reduzir o consumo de energia quando a bateria estiver com um nível baixo de carga, e/ou então para melhorar o desempenho do sistema.

O modelo de interceptadores dinâmicos também oferece a possibilidade de incluir outras propriedades não-funcionais, além das citadas anteriormente, tais como autenticação, encriptação e qualidade de serviço, entre outras. Algumas propriedades não-funcionais podem envolver adaptação dinâmica em múltiplos nós do InteGrade. Um exemplo deste tipo de propriedade não-funcional com possibilidade de implementação pelo modelo de interceptadores é tolerância a falhas através de réplicas.

Esta propriedade não-funcional poderia ser utilizada no InteGrade nos casos onde houvesse falha de execução de uma aplicação, que então seria substituída por uma réplica de forma transparente. Então, ao ocorrer uma falha em um nó, um interceptador poderia substituir o nó que acabou de falhar por uma réplica. Neste caso, esta adaptação envolverá o uso de dois ou mais nós do InteGrade.

No entanto, é importante destacar que as duas propriedades não-funcionais implementadas através do modelo de interceptadores neste trabalho não são exemplos

de adaptação dinâmica em múltiplos nós. Afinal, quando uma destas duas propriedades não-funcionais é ativada pelo modelo de interceptadores dentro do OiL, esta ação vai interferir apenas no LRM do nó da grade que utiliza este ORB. Por exemplo, considere um LRM que está sendo executado dentro de um dispositivo portátil. Se a freqüência com que o LRM envia informações para o *Gerenciador de Aglomerado* for reduzida pelo modelo de interceptadores devido à bateria fraca, esta mudança não vai influenciar um LRM sendo executado num outro dispositivo móvel que está com a bateria carregada.

O mesmo caso vale para a abordagem da OMG para travessia de *firewall* e NAT. Por exemplo, o *Gerenciador de Aglomerado* do InteGrade está dentro de uma rede com *firewall* e NAT. O modelo de interceptadores pode ser utilizado para permitir que um LRM, que está em uma outra rede externa, se comunique com o *Gerenciador de Aglomerado* através de um *proxy* de aplicação. No entanto, um outro LRM, que estivesse dentro da mesma rede que o *Gerenciador de Aglomerado*, pode se conectar diretamente sem a ajuda do modelo de interceptadores.

CAPÍTULO 5

Conclusão

Neste trabalho foi apresentado o modelo de interceptadores dinâmicos que tornou o *middleware* de grade InteGrade capaz de lidar com o dinamismo presente nas grades computacionais através da inclusão de recursos de adaptação dinâmica num dos seus ORBs de comunicação.

A utilização do modelo de interceptadores dinâmicos no OiL possibilitou a este ORB a capacidade de manipular suas propriedades não-funcionais de acordo com as variações do seu ambiente de execução. O InteGrade, aproveitando-se deste recurso de adaptação dinâmica fornecido pelo ORB, consegue agora otimizar dinamicamente sua configuração de acordo com as variações do ambiente de execução das grades computacionais no que diz respeito ao seu comportamento.

Entretanto, apenas os componentes do InteGrade que utilizam o OiL, como por exemplo o LRM, podem usufruir de forma mais direta dos recursos oferecidos pelo modelo de interceptadores. Por isso, para que o InteGrade possa aproveitar totalmente os recursos do modelo de interceptadores dinâmicos, este mecanismo de adaptação dinâmica também deve estar presente no JacORB.

O modelo de interceptadores não inclui mecanismos autonômicos. No entanto, como a decisão sobre a maneira em que a adaptação deve ser feita fica a cargo do usuário, este modelo possui o potencial para ser autonômico.

Para demonstrar os benefícios da utilização deste modelo, foram implementadas duas propriedades não-funcionais. A primeira consiste em desviar as conexões feitas pelos LRMs, que ao invés de serem direcionadas para um nó *Gerenciador de Recursos do Aglomerado* ou então para outros LRMs, são direcionadas para um *proxy*. Através deste *proxy*, os LRMs conseguem se comunicar com o *Gerenciador de Recursos* ou com os outros LRMs mesmo que esses nós estejam presentes num domínio de rede protegido por *firewall* e NAT. Este redirecionamento das conexões é feito pelo modelo de interceptadores apenas quando um arquivo de configuração XML, informando sobre o endereço e porta deste *proxy*, está presente. Se o arquivo não estiver presente, o modelo de interceptadores não realizará qualquer ação e, portanto, os LRMs tentarão se comunicar diretamente com o nó *Gerenciador de Recursos do Aglomerado* ou com os outros LRMs. Desta forma, se

houver necessidade de que estes componentes do InteGrade se comuniquem através de *firewall* e NAT, basta inserir este arquivo de configuração.

Outra opção para detectar se existe a necessidade de redirecionar uma conexão para um *proxy*, poderia ser captar um erro gerado na tentativa de conexão direta do LRM com o *Gerenciador de Recursos* que estivesse dentro de uma rede protegida por *firewall* e NAT. O modelo de interceptadores então tentaria conectar com o *Gerenciador de Recursos* através de um *proxy*. Caso esta tentativa falhasse, o modelo de interceptadores retornaria o erro gerado na tentativa de conexão para o ORB. Com esta abordagem, o modelo de interceptadores seria capaz de decidir se um *proxy* deve ser utilizado de forma autonômica.

A segunda propriedade não-funcional implementada neste trabalho consiste em alterar a freqüência de atualização de informações do LRM para o nó *Gerenciador de Recursos do Aglomerado* com base na porcentagem de utilização da CPU ou no nível de energia da bateria. Por exemplo, se um LRM estiver sendo executado num dispositivo portátil tal como um PDA ou um *laptop*, o modelo de interceptadores poderia monitorar o nível de energia da bateria. Caso a bateria esteja com pouca energia, a freqüência de atualização de informações do LRM seria reduzida pelo modelo de interceptadores. Desta forma, haveria uma economia de consumo.de energia da bateria. Quando a bateria tiver sido recarregada, o modelo de interceptadores deixaria de interferir na freqüência de atualização de informações do LRM, que retornaria ao normal.

Por outro lado, se o LRM estiver sendo executado num computador *Desktop*, o critério para alterar a freqüência de atualização de informações do LRM poderia ser a porcentagem de utilização da CPU. Então, a freqüência de atualização seria reduzida pelo modelo de interceptadores enquanto a utilização da CPU estiver muito alta.

Por isso, diante do que foi apresentado, mostramos que é possível para um *middleware* de grade estático, como o InteGrade, oferecer recursos de adaptação dinâmica sem a necessidade de adicionar novos componentes à sua arquitetura ou modificar os componentes já existentes. A única alteração necessária foi a inclusão do modelo de interceptadores dinâmicos no ORB de comunicação OiL.

A definição da arquitetura do modelo de interceptadores dinâmicos foi feita utilizando o padrão de projeto *Interceptor*, tornando mais simples o seu desenvolvimento. Desta forma, o entendimento da arquitetura deste modelo se torna mais fácil já que foi definida com base em um padrão bem conhecido.

A implementação desta arquitetura foi facilitada pelos recursos oferecidos pela linguagem Lua, utilizada pelo ORB OiL. Por ser uma linguagem fracamente tipada, ou seja, a verificação de tipos de variáveis só ocorre em tempo de execução, uma mesma definição de tabela pôde ser utilizada em diferentes pontos de interceptação os quais recebem diferentes tipos de informações sobre o estado interno do OiL. Outro fator que contribuiu para facilitar a implementação foram as funções de carregamento e ativação de

código em tempo de execução da biblioteca de Lua. Através dessas funções, o modelo de interceptadores dinâmicos consegue ativar dinamicamente as propriedades não-funcionais para responder de forma adequada às variações do ambiente de execução das grades computacionais.

Entretanto, a introdução do modelo de interceptadores no JacORB será mais difícil. Por exemplo, a definição dos pontos de interceptação dentro JacORB será mais complicada de ser feita. Isto se deve ao fato da arquitetura do JacORB ser maior e mais complexa que o OiL. Além disso, o JacORB foi implementado em Java e, portanto as informações do seu estado interno vão estar definidas como atributos dentro de objetos. Isto significa que estas informações só poderão ser obtidas pelo modelo de interceptadores através de métodos destes objetos. Outra diferença se refere ao fato da linguagem Java ser fortemente tipada (ao contrário de Lua), ou seja, uma variável pode armazenar somente um tipo de valor.

Os interceptadores de CORBA, cujo suporte é oferecido pelo JacORB, podem ser uma opção para auxiliar na inserção do modelo de interceptadores no JacORB. Neste caso, uma implementação do modelo de interceptadores dinâmicos seria ativada dinamicamente a partir dos interceptadores de CORBA. No entanto, o modelo de interceptadores teria acesso somente às requisições e respostas transmitidas entre um cliente e o servidor e às referências de um objeto remoto. Portanto, o acesso para outras informações, tais como fluxos de dados recebidos e enviados pelo JacORB, não pode ser feita no JacORB através dos interceptadores de CORBA.

5.1 Resultados Obtidos

A principal contribuição deste trabalho foi demonstrar que é possível ter adaptação dinâmica em uma plataforma de *middleware* de grade sem necessariamente alterá-la, bastando apenas utilizar os serviços de um ORB reflexivo subjacente. Os resultados obtidos deste trabalho são apresentados abaixo:

- Desenvolvimento da arquitetura do modelo de interceptadores dinâmicos que permite a um ORB se adaptar em um ambiente de execução dinâmico através da adição, remoção e ativação dinâmica de propriedades não-funcionais.
- Implementação desta arquitetura no OiL, permitindo ao InteGrade utilizar este ORB como meio para prover suporte a adaptação dinâmica com relação à manipulação de propriedades não-funcionais.
- Implementação de duas propriedades não-funcionais que podem ser utilizadas dinamicamente pelo InteGrade, através do modelo de interceptadores dinâmicos,

quando necessárias para lidar com a dinamicidade e variedade do ambiente de execução das grades computacionais.

5.2 Trabalhos Futuros

Destacamos aqui algumas possibilidades para a continuação deste trabalho:

- Implementação de outras propriedades não-funcionais que contribuam para o InteGrade se adaptar ao dinamismo das grades computacionais oportunistas.
- Inclusão de tratamentos de segurança no modelo de interceptadores. Como foi mostrado neste trabalho, o modelo de interceptadores dinâmicos pode ter acesso a diferentes tipos de informações que trafegam no OiL e, por sua vez, também no InteGrade tais como requisições, fluxos de dados, *sockets* de conexão. Desta forma, existe o risco do modelo de interceptadores ser utilizado como meio para acessar ou adulterar indevidamente informações importantes que estão sendo transmitidas entre os nós do InteGrade através do OiL. Por isso, a inserção de mecanismos de segurança no modelo de interceptadores, tais como autenticação ou criptografia, é necessário.
- Implementação do modelo de interceptadores dinâmicos no JacORB. Desta forma, os componentes que utilizam este ORB também podem aproveitar a capacidade de adaptação provida pelo modelo.
- Implementação do modelo de interceptadores dinâmicos na versão do OiL baseada em componentes.
- Testar até que ponto uma plataforma de *middleware* de grade pode oferecer suporte a adaptação dinâmica utilizando apenas seu ORB de comunicação reflexivo. Por exemplo, verificar se outros mecanismos de adaptação dinâmica, que poderiam ser aplicados diretamente numa plataforma de *middleware* de grade, também teriam condições de serem inseridos em seu ORB de comunicação. Então analisar quais destes mecanismos, que forem inseridos neste ORB de comunicação, ainda permitiriam ao *middleware* de grade oferecer um suporte eficiente para adaptação dinâmica.
- Estudar maneiras para permitir que a decisão pela utilização de propriedades não-funcionais seja feita de forma autonômica pelo modelo de interceptadores dinâmicos.

Referências Bibliográficas

[1] AGARWAL, M; BHAT, V; LIU, H; MATOSSIAN, V; PUTTY, V; SCHMIDT, C; ZHANG, G; ZHEN, L; PARASHAR, M. **Automate: Enabling autonomic grid applications**. Technical Report TR-269, The Applied Software Systems Laboratory, Department of Electrical and Computer Engineering, Rutgers University, 2003.

[2] ARABIE, L; SOETE, G. **Clustering and classification**. World Scientic, 1996.

[3] BERGMANS, L; AKSIT, M. **Aspects and crosscutting in layered middleware systems**. Proceedings of the IFIP/ACM Middleware 2000 Workshop on Reflective Middleware, 2000.

[4] BERMAN, F; FOX, G; HEY, A. J. G; HEY, T. **Grid computing: Making the global infrastructure a reality**. John Wiley & Sons, Inc., 2003.

[5] BLAIR, G; COULSON, G; ANDERSEN, A; BLAIR, L; CLARKE, M; COSTA, F; DURAN-LIMON, H; FITZPATRICK, T; JOHNSON, L; MOREIRA, R; PARLA-VANTZAS, N; SAIKOSKI, K. **The design and implementation of open orb 2**. IEEE Distrib. Syst. Online 2, 6, Setembro 2000.

[6] BUYYA, R. **High Performance Cluster Computing: Architectures and Systems**. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[7] CAMARGO, R. Y; NETTO, M. A. S; ABRANTES, R. L. A. **InteGrade: OO Grid *Middleware***. `http://www.integrade.org.br`, acessado em janeiro de 2008, 2006.

[8] CHINNICI, R; GUDGIN, M; MOREAU, J; SCHLIMMER, J; WEERAWARANA, S. **Web services description language (wsdl) version 2.0 part 1: Core language**. World Wide Web Consortium, Março 2004.

[9] CIRNE, W; SANTOS-NETO, E. **Grids computacionais: Da computação de alto desempenho a serviços sob demanda**. Brazilian Symposium on Computer Networks (SBRC2005), Maio 2005.

[10] CONDOR. **Condor - High Throughput Computing**. `http://www.cs.wisc.edu/condor/`, acessado em janeiro de 2008, 2008.

[11] COSTA, A. T; ENDLER, M; CERQUEIRA, R. **Evaluation of three approaches for corba firewall/nat traversal**. OTM Conferences (2): 923-940, 2005.

[12] COSTA, F; KON, F. **Novas tecnologias de *Middleware*: Rumo à flexibilização e ao dinamismo**. In: JOSé FERREIRA DE REZENDE. (ORG.). MINICURSOS SBRC'002., p. 1–61, Rio de Janeiro, RJ, 2002.

[13] COULOURIS, G; DOLLIMORE, J; KINDBERG, T. **Distributed Systems-Concepts and Design**. Addison Wesley Longman, 2005.

[14] CZAJKOWSKI, K; FERGUSON, D; FOSTER, I; FREY, J; MAQUIRE, S. G. A; SNELLING, D; TUECKE, S. **From open grid services infrastructure to wsresource framework: Refactoring& evolution**. International Business Machines Corporation and The University of Chicago, Maio 2004.

[15] DALY, K. C. **Mean Time Between Flareups, er, Failures**. `http://www.faqs.org/faqs/arch-storage/part2/section-151.html`, acessado em fevereiro de 2008, 2007.

[16] DCOM. **DCOM Technical Overview**. `http://msdn2.microsoft.com/en-us/library/ms809340.aspx`, acessado em janeiro de 2008, 1996.

[17] FOSTER, I; GEISLER, J; NICKLESS, W; SMITH, W; TUECKE, S. **Software infrastructure for the i-way high-performance distributed computing experiment**. In Proceedings of the 5th IEEE Symposium on High Performance Distributed Computing, pages 562-571, 1997.

[18] FOSTER, I; KESSELMAN, C. **Globus: A metacomputing infrastructure toolkit**. International Journal of Supercomputer Applications, 2(11):115-128, 1997.

[19] FOSTER, I; KESSELMAN, C. **The globus toolkit**. In: THE GRID: BLUEPRINT FOR A NEW COMPUTING INFRASTRUCTURE. MORGAN KAUFMANN PUBLISHERS INC., p. 259–278, San Francisco, CA, USA, 1999.

[20] FOSTER, I; KESSELMAN, C. **The grid 2: Blueprint for a new computing infrastructure**. Morgan Kaufmann Publishers Inc., 2003.

[21] FOSTER, I; KESSELMAN, C; TUECKE, S. **The anatomy of the grid: Enabling scalable virtual organizations**. International J. Supercomputer Applications, 15(3), 2001.

[22] FRIEDMAN, R; HADAD, E. **Client-side enhancements using portable interceptors**. Proceedings of the 6th IEEE InternationalWorkshop on Object-oriented Real-time Dependable Systems (WORDS01), Janeiro 2001.

[23] GLOBUS. **The Globus Alliance**. `http://www.globus.org/`, acessado em janeiro de 2008, 2008.

[24] GOLDCHLEGER, A. **Integrade: Um sistema de *Middleware* para computação em grade oportunista**. Master's thesis, Instituto de Matemática e Estatística - Universidade de São Paulo, 2004.

[25] GSI. **Overview of the Grid Security Infrastructure**. `http://www.globus.org/security/overview.html`, acessado em janeiro de 2008, 2008.

[26] GT4. **Globus toolkit 4.0 release manuals**. `http://www.unix.globus.org/toolkit/docs/4.0/`, acessado em janeiro de 2008, 2008.

[27] IERUSALIMSCHY, R. **The Programming Language Lua**. `http://www.lua.org/`, acessado em dezembro de 2007, 2007.

[28] JOHNSON, R. A; WICHERN, D. **Applied multivariate statistical analysis**. Prentice-Hall, 1983.

[29] KON, F; CAMPBELL, R. **Supporting automatic configuration of component-based distributed systems**. In: PROC. 5TH USENIX CONFERENCE ON OBJECT-ORIENTED TECHNOLOGIES AND SYSTEMS (COOTS'99), p. 175–187, San Diego, CA, 1999.

[30] KON, F; CAMPBELL, R. **Dependence management in component-based distributed systems**. IEEE Concurrency, 2000.

[31] KON, F; COSTA, F; BLAIR, G. S; CAMPBELL, R. **The case for reflective middleware: Building middleware that is flexible, reconfigurable, and yet simple to use**. CACM, Vol 45, No 6, 2002.

[32] KON, F; GILL, B; ANAND, M; CAMPBELL, R. H; MICKUNAS, M. D. **Secure dynamic reconfiguration of scalable corba systems with mobile agents**. In: IEEE JOINT SYMPOSIUM ON AGENT SYSTEMS AND APPLICATIONS MOBILE AGENTS, p. 86–98, Zurich, Switzerland, 2000.

[33] KON, F; ROMAN, M; LIU, P; MAO, J; YAMANE, T; MAGALHÃES, L; CAMPBELL, R. **Monitoring, security, and dynamic configuration with dynamictao reflective orb**. In: PROCEEDINGS OF THE IFIP/ACM INTERNATIONAL CON-

FERENCE ON DISTRIBUTED SYSTEMS PLATFORMS AND OPEN DISTRIBUTED PROCESSING (MIDDLEWARE 2000) (PALISADES, NY, APR. 3-7), p. 121–143, Springer-Verlag, Heidelberg, Germany, 2000.

[34] LITZKOW, M; LIVNY, M; MUTKA, M. **Condor - a hunter of idle workstations**. In: PROCEEDINGS OF THE 8TH INTERNATIONAL CONFERENCE OF DISTRIBUTED COMPUTING SYSTEMS, p. 104–111, San Jose, CA, 1988.

[35] LIVNY, M; BASNEY, J; RAMAN, R; TANNENBAUM, T. **Mechanisms for high throughput computing**. SPEEDUP Journal, 11(1), 1997.

[36] MAES, P; NARDI, D. **Issues in computational reflection**. Meta-Level Architectures and Reflection, North-Holland : Elsevier Science Publishers, 1988.

[37] MAIA, R. **Um framework para adaptaçãoo dinâmica de sistemas baseados em componentes distribuídos**. Master's thesis, Departamento de Informática – PUC-Rio, 2004.

[38] MAIA, R; CERQUEIRA, R; KON, F. **A *Middleware* for experimentation on dynamic adaptation**. ARM'05, 2005.

[39] MCKINLEY, P. K; SADJADI, S. M. **Act: An adaptive corba template to support unanticipated adaptation**. Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS04), 2004.

[40] NAHRSTEDT, K; CHU, H; NARAYAN, S. **Qosaware resource management for distributed multimedia applications**. Journal of High-Speed Networking, Special Issue on Multimedia Networking, 7:227-255, 1998.

[41] NOGARA, L. G. **Um estudo sobre middlewares adaptáveis**. Master's thesis, Departamento de Informática – PUC-Rio, 2006.

[42] OGF. **Open Grid Forum**. `http://www.ogf.org/`, acessado em fevereiro de 2008, 2008.

[43] OGSA. **Open Grid Services Architeture**. `http://www.globus.org/ogsa`, acessado em janeiro de 2008, 2008.

[44] OGSI. **Open Grid Services Infrastructure**. `http://www.globus.org/toolkit/draft-ggf-ogsi-gridservice-33_2003-06-27.pdf`, acessado em janeiro de 2008, 2008.

[45] OMG. **Trading Object Service Specification**. Object Management Group - Version 1.0, 2000.

[46] OMG. **Corba firewall traversal specification**. http://www.omg.org/docs/ptc/04-03-01.pdf, acessado em janeiro de 2008, 2004.

[47] OMG. **Name Service Specification**. Object Management Group - Version 1.3, 2004.

[48] OMG. **Common Object Request Broker Architecture: Core Specification**. Object Management Group - Version 3.1, 2008.

[49] RYZIN, J. V. **Classification and clustering**. Academic Press, 1977.

[50] SALLEM, M. A. S; DA SILVA E SILVA, F. J. **Adapta: a framework for dynamic reconfiguration of distributed applications**. In: ARM 06: PROCEEDINGS OF THE 5TH WORKSHOP ON ADAPTIVE AND REFLECTIVE MIDDLEWARE, p. 10, New York, NY, USA, 2006.

[51] SALLEM, M. A. S; DE SOUSA, S. A; DA SILVA E SILVA, F. J. **Autogrid: Towards an autonomic grid middleware**. In: ENABLING TECHNOLOGIES: INFRASTRUCTURE FOR COLLABORATIVE ENTERPRISES, 2007. WET-ICE 2007., p. 223–228, 2007.

[52] SCHMIDT, D; STAL, M; ROHNERT, H; BUSCHMANN, F. **Pattern-Oriented Software Architeture: Patterns for Concurrent and Networked Objects Volume 2**. John Wiley & Sons, Ltd, New York, NY, USA, 2000.

[53] SEIBEL, P. **Practical Common Lisp**. Apress, 2005.

[54] SERVICES, W. **W3C Web Services Activity**. http://www.w3.org/2002/ws/, acessado em janeiro de 2008, 2008.

[55] SPIEGEL, A. **JacOrb-The free Java implementation of the OMG' CORBA standard**. http://www.jacorb.org/documentation.html, acessado em novembro de 2007, 2005.

[56] TANENBAUM, A. S. **Computer networks**. Prentice Hall, 4th edition., 2003.

[57] VALIANT, L. G. **A bridging model for parallel computation**. Communications of ACM, 1990.

APÊNDICE A

Lista de Abreviações

API *Application Programming Interface*

AR *Application Repository*

API *Application Programming Interface*

AR *Application Repository*

ARSM *Application Repository Security Manager*

ARSC *Application Repository Security Client*

ASCT *Application Submission and Control Tool*

BSP *Bulk Synchronous Parallel*

BSPLib *BSP Library*

CCM *CORBA Component Model*

CDRM *Cluster Data Repository Manager*

CkpRep *Checkpoint Repository*

CORBA *Common Object Request Broker Architecture*

CPU *Central Processing Unit*

DSRT *Dynamic Soft Real-Time Scheduler*

DyReS *Dynamic Reconfiguration System*

EM *Execution Manager*

EPS *Event Process System*

GIIS *Grid Index Information Service*

GIOP *General Inter-ORB Protocol*

GRIS *The Grid Resource Information Service*

GRAM *Globus Resource Allocation Manager*

GRM *Global Resource Manager*

GSI *Grid Security Infrastructure*

HTTP *Hypertext Transfer Protocol*

IDL *Interface Definition Language*

IIOP *Internet Inter-ORB Protocol*

IOR *Interoperable Object Reference*

LES *Local Event Service*

LRM *Local Resource Manager*

LuaCCM *Lua CORBA Component Model*

LUPA *Local Usage Pattern Analyzer*

MCT *Minimum Completion Time*

MDS *Monitoring and Discovery Service*

MOP *Meta-Object Protocol*

MPI *Message Passing Interface*

MS *Monitoring Service*

MTBF *Mean Time Between Failures*

NAT *Network Address Translation*

NCC *Node Control Center*

OGSA *Open Grid Services Architecture*

OGSI *Open Grid Services Infrastructure*

OMG *Object Management Group*

OiL *ORB in Lua*

ORB *Object Request Broker*

PDA *Personal digital assistant*

PKI *Public Key Infrastructure*

RFT *Reliable File Transfer*

SSL *Security Socket Layer*

TLS *Transport Layer Security*

WSDL *Web Services Description Language*

WSRF *Web Service Resource Framework*

XML *Extensible Markup Language*