Universidade Federal do Maranhão
Centro de Ciências Exatas e Tecnologia
Programa de Pós-graduação em Engenharia de Eletricidade

Márcio Augusto Sekeff Sallem

Adapta: um arcabouço para o desenvolvimento de aplicações distribuídas adaptativas

São Luís
2007

Márcio Augusto Sekeff Sallem

Adapta: um arcabouço para o desenvolvimento de aplicações distribuídas adaptativas

Dissertação apresentada ao Programa de Pós-graduação em Engenharia de Eletricidade da Universidade Federal do Maranhão, como requisito parcial para a obtenção do grau de MESTRE em Engenharia de Eletricidade.

Orientador: Francisco José da Silva e Silva

Doutor em Ciência da Computação – UFMA

São Luís

2007

Sallem, Márcio Augusto Sekeff

Adapta: um arcabouço para o desenvolvimento de aplicações distribuídas adaptativas / Márcio Augusto Sekeff Sallem. – São Luís, 2007.

116 f.

Dissertação (Mestrado) – Universidade Federal do Maranhão – Programa de Pós-graduação em Engenharia de Eletricidade.

Orientador: Francisco José da Silva e Silva.

1. Programas de Computador. 2. Sistemas adaptativos. 3. Adapta. 4. Middleware. I. Título.

CDU 004.42

Márcio Augusto Sekeff Sallem

Adapta: um arcabouço para o desenvolvimento de aplicações distribuídas adaptativas

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por Márcio Augusto Sekeff Sallem e aprovada pela comissão examinadora.

Aprovada em 14 de dezembro de 2007

BANCA EXAMINADORA

Francisco José da Silva e Silva (orientador)

Doutor em Ciência da Computação – UFMA

Fábio Moreira Costa

Doutor em Ciência da Computação – UFG

Denivaldo Cícero Pavão Lopes

Doutor em Ciência da Computação – UFMA

Aos meus pais, irmãos e minha família.

Resumo

Sistemas computacionais modernos são caracterizados por um alto grau de dinamismo que, junto com a heterogeneidade dos dispositivos computacionais e da infra-estrutura de comunicação, requerem o desenvolvimento de um novo grupo de aplicações capazes de se auto-adaptar dinamica e transparentemente de acordo com mudanças em seu ambiente de execução. Um exemplo destes ambientes é o compreendido por grade computacionais, onde é comum observar alta variabilidade na disponibilidade de recursos, instabilidade de nós, variações no balanceamento de carga e heterogeneidade dos dispositivos computacionais e tecnologia de rede. Outro example são os ambientes de computação móvel, nos quais a grande diversidade de dispositivos computacionais, o dinamismo das redes sem fio, as limitações na disponibilidade de recursos (por exemplo, bateria) e a questão da mobilidade exigem a habilidade do software de se adaptar a mudanças do ambiente.

Este trabalho apresenta o arcabouço Adapta, um middleware reflexivo que provê os meios para desenvolver aplicações distribuídas auto-adaptativas, separando as regras de negócio do código responsável pela adaptação. Adapta também consiste em um sistema em execução que monitora recursos computacionais e notifica componentes da aplicações da importância de eventos que devem disparar ações de reconfiguração. O Adapta define uma linguagem de reconfiguração baseada em XML através da qual é possível especificar como a aplicação deve se adaptar em resposta a mudanças ambientais. Sentenças desta linguagem de reconfiguração podem ser aplicadas em tempo de execução, o que permite alterar dinamicamente o próprio mecanismo de reconfiguração.

Palavras-chaves: Sistemas adaptativos. Reconfiguração dinâmica. Arcabouço. Middleware.

Abstract

Modern computing environments are characterized by a high degree of dynamism that, along with the heterogeneity of computational devices and communication infrastructure, demand the development of a new range of applications that must be able to self-adapt dynamically and transparently according to changes in its execution environment. A computational grid is an example of a modern environment where it is common to notice a high variation on resource availability, node instability, variations on load distribution, and heterogeneity of computational devices and network technology. Another example is mobile computing, where the great diversity of computational devices, wireless network dynamism, limitations regarding available resources (such as, battery power) and mobility require the software to able to adapt to environmental changes.

This paper presents the Adapta framework, a reflective middleware that provides the means to develop self-adaptive distributed applications, separating the business code from the one responsible for adaptation. Adapta also provides a runtime execution environment that monitors computational resources and notifies application components about the occurrence of important events that should trigger reconfiguration actions. Adapta provides a XML based reconfiguration language that defines how the application must adapt in response to environmental changes. Statements of the reconfiguration language can also be applied at runtime, which allows to dynamically change the reconfiguration mechanism itself.

Keywords: Adaptive systemas. Dynamic reconfiguration. Framework. Middleware.

Agradecimentos

À Deus, pela existência e pela inteligência. Aos meus pais, João e Arlete, pela fé e dedicação em seus filhos. A meus irmãos Kércio, Flávio e Bianka que sempre se fizeram presentes em toda minha vida. A minha família pelo carinho e admiração.

Ao meu orientador Francisco, pelo acompanhamento sempre presente, pela paciência e momentos despendidos e pelas lições e experiências compartilhadas que sem dúvidas contribuíram na minha formação acadêmica e pessoal.

A Stanley Araujo, Gilberto Cunha, Rafael Fernandes e Paulo José, membros do Laboratório de Sistemas Distribuídos, pela ajuda e envolvimento neste projeto.

"Deus não me deu asas, mas subindo degraus todos os dias chegarei ao céu."

Autor desconhecido

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Símbolos

AOM	*Adaptive Object Model*
ARM	*Application Replication Manager*
$CASA$	*Contract-based Adaptive Software Architecture*
CEL	*Composite Events Language*
$DRACO$	*Distrinet Reliable and Adaptive COmponents*
EM	*Execution Manager*
EPS	*Event Processing System*
GPS	*Global Position System*
GRM	*Global Resource Manager*
LEL	*Local Events Language*
MCT	*Minimun Completion Time*
MET	*Minimun Execution Time*
ML	*Monitoring Language*
MVC	*Model View Controller*
$MTBF$	*Mean-Time Between Failure*
PDA	*Personal Digital Assistant*
POA	Programação Orientada a Aspectos
RL	*Reconfiguration Language*
RMI	*Remote Method Invocation*
SA	*Switching Algorithm*

1 Introdução

Uma das características dos sistemas computacionais modernos é a grande heterogeneidade dos dispositivos computacionais. Este aspecto tem afetado o desenvolvimento de novas aplicações na medida em que estas se voltam para um conjunto mais amplo de plataformas computacionais, que variam desde computadores de alta velocidade a dispositivos portáteis com recursos limitados (por exemplo, telefones celulares e PDAs). As aplicações que executam nestes ambientes heterogêneos devem ser flexíveis, ou seja, capazes de executar em diversas plataformas computacionais, desde que atendendo a um conjunto mínimo de pré-requisitos.

Outra característica relevante de sistemas computacionais modernos é o alto grau de dinamismo, compreendendo modificações abruptas e imprevisíveis na disponibilidade de recursos do ambiente de execução. Em muitos casos, essas modificações podem prejudicar a qualidade de serviço, ou ainda interromper temporariamente a aplicação. Em face deste dinamismo, as aplicações devem ser capazes de modificar sua estrutura ou seu funcionamento, adaptando-se a mudanças observadas no ambiente de execução.

Um exemplo de um ambiente computacional moderno, heterogêneo e dinâmico, é aquele apresentado pela computação móvel. Esse ambiente caracteriza-se pela variação na disponibilidade de recursos e serviços, conectividade intermitente, alterações na largura de banda e nas taxas de erros de transmissão, alta heterogeneidade de dispositivos e de tecnologias de comunicação, além de questões envolvendo a mobilidade dos usuários.

Outro exemplo é uma grade computacional oportunista, na qual o tempo ocioso de recursos distribuídos é utilizado para a realização de computações complexas. É comum observar-se uma alta variação na disponibilidade de recursos, grande instabilidade dos nós em decorrência de não serem dedicados nem constituírem um ambiente isolado, variações na distribuição de carga entre os nós envolvidos, grande heterogeneidade de recursos computacionais e de tecnologias de rede interligando os nós.

Como forma de mitigar os problemas decorrentes do dinamismo observado em ambientes computacionais modernos, o software deve ser capaz de alterar seu comportamento dinamicamente em decorrência de mudanças em seu ambiente de execução. Por

exemplo: uma aplicação de vigilância que transmite vídeos em tempo real para uma central pode reduzir a resolução do vídeo transmitido caso haja uma diminuição da largura de banda disponível. Outro exemplo é uma aplicação de gerenciamento de processos em ambientes distribuídos, que pode migrar serviços de um nó computacional que se encontra sobrecarregado para outros nós que estejam ociosos, melhorando o balanceamento de carga do ambiente. As aplicações capazes de se reconfigurar automaticamente em face de mudanças ocorridas no ambiente computacional são chamadas genericamente de software auto-adaptativo.

1.1 Desafios

Software auto-adaptativo introduz um novo grau de complexidade para o desenvolvedor, pois além de projetar o comportamento funcional da aplicação, ele deve se preocupar com questões relacionadas à adaptação em si, como por exemplo: quais aspectos do ambiente de execução devem ser monitorados; como realizar o monitoramento; como detectar mudanças relevantes no ambiente de execução; quais adaptações de software devem ser efetivadas; e em qual momento isto deve ocorrer. O desenvolvedor deverá escolher entre diversos mecanismos de adaptação, desde a simples modificação de parâmetros da aplicação à reorganização dos componentes que a compõem. Este processo de adaptação pode ser feito na própria aplicação ou através de um middleware específico. Em ambos os casos, o código funcional da aplicação deve ser desacoplado do código responsável pela adaptação, de forma a minimizar sua complexidade e facilitar a manutenabilidade.

Um dos elementos que compõem um software auto-adaptativo é a infra-estrutura de monitoramento dos recursos computacionais, responsável pela observação do ambiente de execução a fim de que sejam detectadas mudanças que ensejem alterações no comportamento da adaptação. Esta atividade compreende alguns desafios, tais como: a diversidade de plataformas computacionais com características próprias e interfaces substancialmente distintas umas das outras; quais recursos computacionais irão ser monitorados; e como monitorar recursos não conhecidos a priori sem interromper o serviço.

Outro aspecto relevante envolve a detecção e notificação de mudanças significativas no ambiente computacional, capazes de provocar alterações na estrutura ou no funcionamento de uma aplicação. A definição do que seja uma mudança significativa é es-

pecífica do domínio da aplicação e variável com o tempo. Conseqüentemente, um software auto-adaptativo deve conter meios para expressar essas mudanças em tempo de execução. Além disso, a detecção de uma mudança deve ser tomada com cuidado, especialmente em ambientes distribuídos, buscando minimizar a quantidade de mensagens de notificação enviadas. Este cuidado deve ser redobrado em nós com baixo poder de processamento e alimentados por uma fonte de energia não-renovável (baterias), a fim de evitar o consumo excessivo de energia.

Um último aspecto corresponde aos mecanismos usados para a adaptação dos componentes da aplicação. O desenvolvedor de software auto-adaptativo deve decidir, entre diversas formas de reconfiguração, aquela mais adequada ao domínio e o estado de execução atual da aplicação. É importante que o mecanismo de reconfiguração seja independente do aspecto a ser adaptado (por exemplo, requisito funcional e não-funcional) e extensível, possibilitando a inserção dinâmica de outros mecanismos não previstos em tempo de compilação.

1.2 Objetivos

Esta pesquisa tem por objetivo geral o desenvolvimento de um arcabouço genérico e flexível para o desenvolvimento de aplicações distribuídas adaptativas, a partir da integração das abordagens [7], [66] e [28]. Os objetivos específicos deste trabalho compreendem:

- Estudo do estado da arte em sistemas de reconfiguração dinâmica e arcabouços para criação de aplicações adaptativas;
- Definição de uma arquitetura para o arcabouço composta por uma infra-estrutura de monitoramento do ambiente distribuído, um serviço para notificação de eventos e um componente para reconfiguração dinâmica da aplicação;
- Implementação da arquitetura proposta;
- Avaliação do arcabouço, a ser constituída de medições que analisem a aderência aos requisitos estabelecidos na seção 3.2 e sua efetividade na criação de aplicações distribuídas adaptativas.

1.3 Organização

Esta dissertação está organizada da seguinte forma. No Capítulo 2, encontra-se a fundamentação teórica de software auto-adaptativo. Serão apresentadas taxonomias para classificação de sistemas de software auto-adaptativo, as tecnologias e os princípios que permitem o desenvolvimento destes sistemas e as áreas de uso, com ênfase em grades computacionais e computação móvel.

No Capítulo 3, será exibida uma visão geral do arcabouço, seus componentes e interações. Para cada componente serão descritas as principais funcionalidades, a sua interface e a linguagem de reconfiguração, bem como justificadas as decisões de implementação tomadas.

No Capítulo 4, realizamos uma avaliação do arcabouço proposto, através da análise de dois estudos de caso: a introdução de mecanismos autônomos ao middleware de grade Integrade e a inserção dinâmica de um algoritmo de compressão em um servidor de vídeo, tornando-o adaptivo com relação à variações na largura de banda disponível para a comunicação com seus clientes. São descritos experimentos e simulações que quantificam os benefícios da introdução de comportamento adaptivo nos sistemas utilizados nestes estudos de caso.

O Capítulo 5, apresenta trabalhos relevantes relacionados ao desenvolvimento de aplicações adaptativas e arcabouços para adaptação dinâmica das aplicações, comparando estas abordagens ao arcabouço proposto neste trabalho. O Capítulo 6 apresenta as conclusões obtidas e as perspectivas de trabalhos futuros decorrentes da pesquisa realizada.

2 Sofwtare Auto-Adaptativo

Software auto-adaptativo é aquele capaz de reconfigurar suas operações de forma a aperfeiçoar o seu funcionamento ou se recuperar de uma falha ou mesmo incorporar funcionalidades adicionais. Isso somente é possível se o software monitorar seu ambiente de execução e avaliar seu comportamento e desempenho em tempo de execução, determinando em seguida o que, quando e como se reconfigurar.

Este capítulo apresenta uma introdução acerca de softwares adaptativos. Na Seção 2.1 são apresentadas algumas definições iniciais; a Seção 2.2 enumera três taxonomias que permitem posicionar, classificar e comparar sistemas e tecnologias de software auto-adaptativo; a Seção 2.3 ilustra princípios e tecnologias que permitem o desenvolvimento sistemático de software auto-adaptativo; e finalmente, a Seção 2.4 aborda áreas da computação em que o uso de software auto-adaptativo é promissor, por exemplo, em grades computacionais oportunistas e na computação móvel.

2.1 Definições Iniciais

Oreizy [43] conceitua software auto-adaptativo como *aquele que modifica seu comportamento em resposta a mudanças em seu ambiente operacional, este compreendendo qualquer elemento monitorável pelo sistema, como entrada do usuário, dispositivos de hardware e sensores.*

De forma análoga, Paul Robertson et all [52] definem software auto-adaptativo *como aquele que avalia o seu próprio comportamento e o modifica quando o resultado dessa avaliação indica que o software não está cumprindo o seu papel ou quando é possível obter melhor desempenho ou melhor funcionalidade.*

Robert Laddaga et all [29] complementam as definições apresentadas observando que os software auto-adaptativo *tem múltiplas formas de atingir seu objetivo, além de conhecimento suficiente de sua estrutura para fazer mudanças efetivas em tempo de execução.*

2.2 Taxonomia de Software Auto-Adaptativo

Uma taxonomia de software auto-adaptativo é importante porque permite posicionar o estado atual das ferramentas e técnicas dentro deste domínio, provendo os meios para compará-las e combiná-las.

2.2.1 Taxonomia de Satyanarayanan

Uma das mais importantes taxonomias de adaptação para software é a de Satyanarayanan [56], que consiste na delimitação de dois extremos. Em um dos extremos, a aplicação é totalmente responsável pela adaptação. Esta abordagem, denominada *laissez-faire*, independe de suporte do sistema, mas é prejudicada pela ausência de um ponto central responsável por resolver conflitos de alocação de recursos e assegurar limites de uso dos recursos. Além disso, as aplicações tornam-se mais difíceis de ser escritas porque o desenvolvedor deverá se preocupar com o código responsável pela adaptação.

No outro extremo, tem-se que a adaptação é de responsabilidade do sistema e transparente às aplicações, sem envolver alteração no código destas. Esta abordagem é denominada *application-transparent*. Nela, o sistema representa o ponto central de controle e gerenciamento de recursos. Em razão da adaptação efetuada pelo sistema ser genérica, em muitos casos ela se mostra inadequada ou improdutiva à aplicação.

Entre os extremos existe um espectro de possibilidades de adaptação chamado *application-aware*. Nestes casos, ocorre uma colaboração entre a aplicação e o sistema que permite que aplicações determinem como melhor se adaptar, mas preservando a capacidade do sistema de monitorar o ambiente computacional e assegurar a decisão de alocação de recursos entre as aplicações.

2.2.2 Taxonomia de Mens, Buckley, Zenger e Rashid

Mens, Buckley, Zenger e Rashid [60] abordam a evolução de software sob o prisma dos mecanismos e ferramentas disponíveis para o suporte a mudanças no software. Essa análise permite caracterizar os mecanismos de mudança de software e classificar os fatores que influenciam esses mecanismos. Diante disso, é apresentada uma taxonomia que compreende quatro grupos de aspectos: as propriedades temporais, os objetos sujeitos a mudança, as

propriedades do sistema e os mecanismos de suporte.

As **propriedades temporais** referem-se ao tempo, ao histórico e à freqüência da mudança. O tempo da mudança consiste na fase no ciclo de vida do software em que se dá a alteração, podendo ser: em tempo de compilação, quando as mudanças ocorrem em nível de código fonte e exigem re-compilação; em tempo de carga, quando os elementos são carregados dentro do executável do software no momento de sua inicialização; e em tempo de execução, quando o software adiciona, remove ou substitui os seus componentes e modifica propriedades em tempo de execução, sem necessidade de interrupção do serviço. O histórico de mudança consiste no controle e gerenciamento das versões do software, podendo ser: sem suporte a versões, onde novas versões do software sobrepõem versões antigas de forma destrutiva; com suporte estático a versões, na qual versões novas e antigas coexistem somente em tempo de compilação e carga e não podem ser usadas simultaneamente em tempo de execução; e com suporte dinâmico a versões, que permite que versões novas e antigas coexistam em tempo de execução. A freqüência da mudança indica a periodicidade em que mudanças ocorrem no software, podendo ser: continuamente, periodicamente ou em intervalos arbitrários.

Os **objetos sujeitos a mudanças** compreendem o artefato, a granularidade, o impacto e a propagação da mudança. O artefato sujeito a mudança consiste em um dos elementos que formam o software, tais como arquitetura, código-fonte, módulos, componentes, etc. A granularidade consiste no nível de detalhamento da mudança, por exemplo: a mudança de um componente tem granularidade espessa, enquanto a alteração de um parâmetro, granularidade fina. O impacto classifica a mudança de acordo com o seu impacto no software, podendo ser: restritas localmente a um elemento do software ou globais a todo o espaço do software. A propagação avalia se a mudança efetuada em um elemento do software precisa ser coordenada ou sincronizada com outro elemento do software.

As **propriedades do sistema** incluem disponibilidade, atividade, abertura e segurança. A disponibilidade distingue software de alta e baixa disponibilidade. A atividade classifca software em: reativo cuja mudança é conduzida por impulso externo, e pró-ativo onde a mudança ocorre de forma automática a partir do monitoramento do ambiente de execução. A abertura indica se um software é construído ou não para aceitar extensões e permite distinguir software fechado, que aceita novas extensões apenas no código fonte antes da compilação, de software aberto, que é desenvolvido especificamente para facilitar a incorporação de novas extensões, estática ou dinamicamente. A

segurança é a propriedade que indica se a mudança não prejudicará o comportamento e funcionamento do sistema.

Os **mecanismos de suporte** referem-se ao grau de automação e de formalismo e ao tipo de mudança. O grau de automação compreende mudanças automáticas, parcialmente automáticas e manuais, enquanto o grau de formalismo classifica a implementação na forma *ad-hoc* ou baseada em algum formalismo matemático. Quanto ao escopo do tipo de mudança taxonomia restringe-se apenas a: mudança estrutural, quando a esta altera os elementos que compõem o software e, possivelmente, o comportamento do software, e mudança semântica, quando modifica os requisitos e objetivos de um software.

2.2.3 Taxonomia de McKinley, Sadjadi, Kasten and Cheng

McKinley, Sadjadi, Kasten and Cheng [35] distinguem duas formas de se realizar adaptação dinâmica de software. Uma delas é a adaptação de parâmetros, que envolve a modificação de variáveis para determinar o comportamento do software. A outra forma é a adaptação composicional que se baseia na substituição dinâmica de algoritmos e componentes do software para se adaptar ao ambiente de execução. Nesta forma de adaptação, é possível classificar software auto-adaptativo a partir da resposta às seguintes perguntas:

Como? Refere-se aos mecanismos usados para possibilitar a adaptação composicional de software. Por exemplo: programação orientada a aspectos e reflexão computacional. Em geral, os mecanismos criam um nível de indireção entre as interações dos elementos que compõem o software. Uma entidade, denominada iniciadora, usa essas técnicas para adaptar uma aplicação. Essa entidade iniciadora pode ser humana (desenvolvedor de software ou administrador do sistema) ou um elemento de software (meta-objeto ou sistema de suporte a execução). Nessa dimensão, pode-se classificar ainda os graus de transparência da abordagem adotada, que determinam a portabilidade da solução e o grau de dificuldade de adicionar comportamento adaptativo em programas existentes. Por exemplo, diz-se que uma abordagem de middleware é transparente ao código da aplicação se o desenvolvedor não precisar modificar o código da mesma para torná-la adaptativa.

Quando? Indica o momento em que o código adaptativo é composto ao código responsável pelas regras de negócio. Quanto mais tarde ocorrer o processo de composição, mais poderosos serão os mecanismos de adaptação, porém será mais difícil assegurar a consistência da aplicação. Existem dois tipos de composição, a estática e a dinâmica. No

primeiro tipo, a composição da aplicação dá-se em tempo de desenvolvimento, compilação ou carga. Por exemplo: a introdução ou remoção dinâmica de aspectos durante a compilação ou a carga dinâmica de binários na máquina virtual Java (JVM). Por outro lado, a composição dinâmica é o mecanismo mais flexível porque ocorre em tempo de execução. Assim, algoritmos e componentes podem ser substituídos ou estendidos durante a execução sem interrupção ou reinício da aplicação. Por exemplo: a adição dinâmica de um algoritmo de criptografia em redes sem segurança.

Onde? Indica o local no sistema em que o código adaptativo é inserido. As possibilidades incluem uma das camadas de middleware ou o código da aplicação. Quando a adaptação se dá na camada de infra-estrutura de middleware, ela pode envolver a construção de uma camada de serviços de comunicação adaptáveis ou a extensão de uma máquina virtual para facilitar a interceptação e o redirecionamento de interações no código funcional da aplicação. Outras soluções introduzem código adaptativo na camada de distribuição e de serviços comuns do middleware e se baseiam na interceptação de mensagens associadas a invocações de métodos remotos e o processamento dessas mensagens de acordo com o estado da aplicação e as condições do ambiente de execução.

2.3 Abordagens para Desenvolvimento de Software Auto-Adaptativo

No tocante ao desenvolvimento de software auto-adaptativo, alguns princípios e tecnologias vêm sendo utilizados para facilitar esse processo. Esta seção apresentará algumas dessas abordagens.

2.3.1 Separação de Responsabilidades

O princípio da separação de responsabilidades consiste na identificação de características (responsabilidades) e na separação do problema em pedaços menores que correspondem a cada uma das responsabilidades. Por exemplo, o padrão de desenvolvimento *Model-View-Controller* (MVC) [6] separa o desenvolvimento da aplicação em três camadas: o **Modelo**, camada de dados que implementa a lógica do software; a **Visão**, camada de apresentação e interação com o usuário final; e o **Controle**, camada responsável pelo fluxo

da aplicação. Desta forma, o MVC separa o conteúdo da apresentação e o processamento dos dados. As camadas não se sobrepõem em funcionalidade, mas colaboram umas com as outras para atingir seu objetivo.

No desenvolvimento de software auto-adaptativo, a separação de responsabilidades permite que o código funcional da aplicação (responsável pelas regras de negócio) seja separado do código responsável pela adaptação. Isto simplifica o desenvolvimento, a manutenção e o reuso do código adaptativo.

2.3.2 Desenvolvimento Orientado a Componentes

O desenvolvimento orientado a componentes [21] permite aos desenvolvedores de software reutilizar componentes, muitas vezes escritos por terceiros, para construir sistemas de software maiores e complexos. Isto somente é possível se os componentes forem definidos através de interfaces que agem como contratos entre eles. Além do alto grau de reutilização, o uso de componentes facilita a manutenção de software e torna o seu desenvolvimento mais ágil e rápido.

O uso de componentes permite a adoção de mecanismos de reconfiguração estrutural em software auto-adaptativo. Esses mecanismos referem-se à adição, remoção e substituição de componentes em tempo de execução, em face a mudanças no ambiente computacional. Por exemplo, um software de visualização de atividades vulcânicas pode substituir um componente coletor de vídeos por outro coletor de imagens, em razão da degradação da qualidade no enlace de rede.

2.3.3 Reflexão Computacional

Reflexão computacional [33] é a habilidade de um software observar ou até mesmo modificar a sua estrutura ou comportamento. Essa abordagem expõe detalhes de implementação do software em um nível de abstração que permite mudanças em seu comportamento sem comprometer a sua portabilidade. Dessa maneira, software reflexivo deve incorporar estruturas de dados que representam os diversos aspectos do software, em uma auto-representação. Esses aspectos são causalmente conectados com os aspectos de implementação do sistema. Portanto, modificações em qualquer um desses aspectos levam a mudanças no outro aspecto. Isto assegura que o software sempre tem uma auto-

representação precisa de si mesmo e que o estado e a computação do software estarão em conformidade com essa representação.

A reflexão computacional compreende duas atividades: introspecção e intercessão [59]. A introspecção denota a capacidade que um software tem de examinar sua própria estrutura, estado e representação. A esses elementos dá-se o nome de meta-informação, que representa qualquer informação contida e manipulável por um software computacional que seja referente a si próprio. Por exemplo: um software pode observar quais os métodos que compõem uma interface, os parâmetros e seus tipos de dados.

A outra atividade da reflexão é a intercessão que permite que um software aja sobre as observações obtidas e modifique seu estado e comportamento. Isto é realizado a partir da interceptação de operações realizadas pelo software e posterior modificação dessas. Por exemplo: quando o software invocar uma chamada a um escalonador de processos, esta será interceptada e modificada, introduzindo um código mais adequado a situação do ambiente computacional.

Ferber [17] distingue dois modelos de reflexão computacional. O primeiro modelo chama-se reflexão estrutural, no qual cada entidade é uma instância de uma classe, e cada classe é uma intância de uma outra classe chamada meta-classe. Nesse modelo, é possível estender a parte estrutural dos objetos, por exemplo, modificando os tipos de dados de seus atributos. O segundo modelo é chamado de reflexão comportamental, que é baseado no uso de reflexão no gerenciamento de mensagens, dessa maneira modificando o comportamento dos objetos.

Em geral, software reflexivo é desenvolvido usando-se o padrão de desenvolvimento *reflection* [6]. Esse padrão divide um software em um meta-nível e em um nível base. O meta-nível representa a estrutura e o comportamento do software em elementos chamados meta-objetos; enquanto o nível base define a lógica da aplicação e a implementação das regras de negócio. Esses dois níveis são causalmente conectados através de um protocolo de meta-objetos [26], de forma que modificações em qualquer um dos dois serão refletidas no outro. Esse protocolo de meta-objetos ainda permite introspecção e intercessão de forma sistemática (ao invés de *ad hoc*) nos objetos do nível base.

A principal vantagem da organização do software reflexivo em duas camadas é a nítida separação do código em dois níveis de funcionalidade, um nível base que provê funcionalidade do domínio e um meta-nível que permite que o comportamento, a forma ou

a implementação do nível base sejam manipulados, regulados ou influenciados [46]. Em decorrência disto, software reflexivo apresenta outras vantagens como maior reutilização do código, mais facilidade na manutenção e depuração do código e maior transparência na incorporação de conteúdo adaptativo no software.

2.3.4 Programação Orientada a Aspectos (POA)

Kiczales et all [27] observou que em alguns sistemas de software complexos, desenvolvidos segundo o paradigma da orientação a objeto, existem vários requisitos funcionais e não-funcionais entrelaçadas ao longo do código computacional, tais como: padrões de acesso à memória, tolerância a falhas, segurança e persistência de dados. Isto dificulta o desenvolvimento e manutenção do software, pois a alteração em um desses requisitos implica na modificação de todas as classes que utilizam o mesmo.

A programação orientada a aspectos (POA) surgiu como consequência do princípio de separação de responsabilidades e permite expressar responsabilidades entrelaçadas em termos de elementos chamados aspectos, desenvolvidos separadamente de outras partes do sistema e armazenados em uma unidade de código visível a todos os componentes do sistema [64]. Desta maneira, o software é organizado em classes ou componentes que representam as regras de negócio da aplicação (regras de negócio), e aspectos relacionam os requisitos funcionais e não-funcionais que afetam o comportamento do sistema e que estariam entrelaçados no código das classes e componentes. Além de aspectos, POA introduz outros conceitos:

- **Pontos de junção ou *join points***, que são pontos na execução de um software em que se pode inserir códigos de aspecto, por exemplo: chamada e execução de métodos, inicializacação de construtores, execução de tratamento de exceções, dentre outros;

- **Conjuntos de junção ou *pointcuts***, que consistem em uma especificação de um subconjunto de pontos de junção, determinado através de uma expressão lógica usando operadores booleanos. Por exemplo: um *pointcut* pode definir um subconjunto de todos os métodos cujo nome seja iniciado pela expressão *get*, para introduzir o código do aspecto;

- **Adendos ou *advices***, que são comportamentos adicionais complementares à exe-

cução do aspecto. Eles podem ser executados antes, depois ou enquanto o código do ponto de junção é executado. Por exemplo, um *advice* pode ser usado para verificar se ocorreu alteração significativa na disponibilidade de um recurso computacional antes de ser executado o código do aspecto.

O uso de POA auxilia o desenvolvimento de software auto-adaptativo, haja visto que muitas mudanças no ambiente computacional estão relacionadas a requisitos entrelaçados no código da aplicação. Dessa maneira, o encapsulamento desses requisitos em aspectos permite a modificação do software em um único ponto, ao invés de modificar todos os locais em que esse requisito está presente. Além disso, é possível substituir dinamicamente a implementação de um aspecto por outra sem que haja modificação no código da aplicação. Por exemplo: a modificação do mecanismo de persistência de dados utilizado pela aplicação em face da diminuição da largura de banda disponível.

POA realiza a introdução dos aspectos nos respectivos pontos no código da aplicação em tempo de compilação. Isto gera um código entrelaçado que não pode ser tornado reconfigurável com facilidade. Para superar esta dificuldade, em alguns projetos como o AspectWerkz [4] e o PROSE [41] foi proposto o conceito de POA dinâmico, que é a capacidade de introduzir dinamicamente os aspectos na aplicação, permitindo a adição, remoção ou substituição de aspectos em tempo de execução.

Rashid e Kortuem [50] vêem a própria adaptabilidade da aplicação como um aspecto genérico. Desta maneira, POA pode ser visto como uma forma de modularizar a adaptação de aplicações. Por exemplo, as telas dos dispositivos computacionais em um ambiente pervasivo variam em relação às suas propriedades, como suporte a imagens, quantidade de cores e resolução. Nesses casos, uma aplicação pode realizar a adaptação do conteúdo exibido em um aspecto, que verifica o suporte do dispositivo a algumas propriedades básicas de visualização e modifica o conteúdo dinamicamente. Se combinado com POA dinâmico, o aspecto que contém o comportamento adaptativo pode ser adicionado ou removido dinamicamente, alterando o próprio mecanismo de adaptação da aplicação.

2.3.5 Middleware

O termo middleware apareceu pela primeira vez no final dos anos 80 para descrever software de gerenciamento de conexões de rede. A popularização do termo deu-se apenas

em meados dos anos 90, quando a tecnologia de rede adquiriu maior visibilidade e penetração no mercado. A esta altura, o middleware evoluiu para um conjunto mais rico de paradigmas e serviços oferecidos para tornar mais fácil e gerenciável o desenvolvimento de aplicações distribuídas [3].

O middleware é uma tecnologia de software desenvolvida para gerenciar a complexidade e heterogeneidade inerentes a sistemas distribuídos. A camada do middleware localiza-se logo acima da camada do sistema operacional e abaixo da camada da aplicação e oferece ao desenvolvedor de sistemas uma camada de abstração ao longo de todo o sistema distribuído. Por exemplo, CORBA é um middleware que assegura a interoperabilidade entre diferentes plataformas de hardware e sistemas operacionais, com uma série de serviços para aplicações em um ambiente distribuído, como serviço de nomes, de persistência, dentre outros.

Schantz e Schmidt [57] organizaram o middleware em quatro camadas, ilustradas na Figura 2.1. Uma camada de infra-estrutura do nó encontra-se logo acima do sistema operacional e provê uma interface que abstrai a heterogeneidade dos dispositivos computacionais, sistemas operacionais e protocolos de rede. Em seguida, temos a camada de distribuição, que define abstrações de programação distribuída de alto-nível (por exemplo, objetos remotos), que permitem que os desenvolvedores escrevam aplicações distribuídas de forma similar a aplicações centralizadas. Nesta camada, encontram-se plataformas de middleware de interoperabilidade, como o CORBA e Java RMI. Uma camada de serviços comuns que inclui suporte a tolerância a falhas, segurança, persistência e transações. Finalmente, tem-se a camada específica do domínio, que é direcionada para requisitos dos vários domínios de aplicação, como automação de processos, comércio eletrônico, dentre outros.

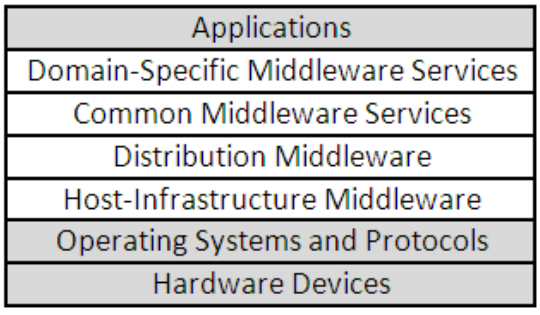

Figura 2.1: Camadas de Middleware

Em razão de suas características, middleware tem se tornado um ponto estra-

tégico para a inserção de comportamento adaptativo relativo a aspectos do sistema como, por exemplo, qualidade de serviço, segurança e tolerância a falha, dentre outros. Em geral, as técnicas de adaptação em middleware envolvem interceptação e modificação de mensagens em uma camada de indireção. Essa camada oferece um grau de transparência ao processo de adaptação. Desta maneira, os desenvolvedores não precisam alterar o código das aplicações para que as mesmas apresentem adaptabilidade, haja visto que toda a lógica da adaptação encontra-se no middleware. A desvantagem de se posicionar os mecanismos adaptativos no middleware decorre de que certas adaptações necessitam de informações que apenas estão disponíveis na camada de aplicação. Por exemplo: a alteração da taxa de envio de quadros em um servidor de vídeo.

2.4 Áreas de Uso

A motivação para o desenvolvimento de software auto-adaptativo está relacionada com o surgimento de ambientes computacionais caracterizados por grande heterogeneidade de dispositivos computacionais e da infra-estrutura de comunicação, além de acentuado dinamismo no ambiente computacional. Esta seção apresenta a importância do software auto-adaptativo em dois ambientes computacionais de inegável relevância e que têm experimentado acentuado desenvolvimento nos últimos anos: as grades de computadores e ambientes de computação móvel.

2.4.1 Grade Computacional

Com a evolução da infra-estrutura de comunicação, houve o surgimento de novas formas de interação entre os diversos dispositivos computacionais que compõem um sistema distribuído. Além do compartilhamento de arquivos e serviços, outros recursos computacionais como ciclos do processador e espaço de armazenamento puderam ser compartilhados para atingir um fim comum, como por exemplo, o processamento de uma aplicação.

Este novo modelo de interação não se restringe a um mesmo domínio administrativo e pode se expandir geograficamente pelo mundo todo, envolvendo diversos domínios administrativos. Um middleware é necessário para tornar toda esta infra-estrutura transparente ao usuário final, apresentando a computação distribuída e os recursos computacionais como uma entidade única, chamada de grade computacional [11, 18].

Grades computacionais estimularam a concepção de uma nova geração de aplicações que permitem a combinação de computações, experimentos, observações e dados obtidos em tempo real. Os fenômenos modelados por essas aplicações exigem componentes de software cuja composição e interação são extremamente dinâmicas. Além disso, a infra-estrutura de grade é heterogênea e dinâmica, agregando uma grande quantidade de recursos computacionais e meios de comunicação, bancos de dados e muitas vezes, sensores e periféricos específicos. O dinamismo em grades pode ser observado na alta variação da disponibilidade de recursos, na instabilidade dos nós, em variações da carga de trabalho nos nós e enlaces de rede, dentre outros fatores.

Grades computacionais oportunistas [20] têm um dinamismo ainda mais acentuado. Estas grades utilizam os recursos computacionais ociosos oferecidos por máquinas de usuários comuns para execução de aplicações paralelas. Durante o ciclo de execução da aplicação, um usuário pode executar uma aplicação local que consuma todo o poder computacional, ou ainda desligar a máquina sem aviso. Desta forma, a natureza dinâmica da infra-estrutura de grade, sua alta escalabilidade e grande heterogeneidade tornaram impraticável sua configuração, manutenção e recuperação (em caso de falhas) unicamente por seres humanos.

Muitos projetos de pesquisa recentes, como o AutoMate [45] e o OptimalGrid [25], reconheceram a necessidade de prover maior autonomia às grades computacionais, um dos grandes desafios para esta nova geração de sistemas. O termo computação autônoma tem sido usado para descrever um sistema que exibe propriedades tais como: auto-cura, que é a habilidade de um sistema de ser ciente de possíveis problemas, detectar eventuais falhas e reconfigurar-se de forma a manter o funcionamento normal; e auto-otimização, que é a habilidade de detectar a degradação de desempenho e inteligentemente executar ações de otimização.

Diante disso, uma grade que apresenta auto-cura poderia decidir dinamicamente qual o algoritmo de recuperação de falhas mais adequado (replicação ou pontos de controle) em resposta a mudanças no ambiente computacional. A própria técnica de recuperação pode ser adaptável. Por exemplo, ao adotar o uso de replicação, pode-se ajustar dinamicamente a quantidade de réplicas a serem geradas para cada submissão de aplicações.

Ao mesmo tempo, uma grade que apresenta auto-otimização poderia ser capaz

de selecionar, dinamicamente, o algoritmo de escalonamento mais apropriado ao ambiente computacional, usando parâmetros globais da grade como a taxa de chegada de aplicações e o percentual de uso de recursos. Além disso, técnicas de re-escalonamento com balanceamento de carga dinâmico permitem melhor desempenho da grade, com maximização do uso dos recursos computacionais.

2.4.2 Computação Móvel

A evolução da infra-estrutura de comunicação sem fio, com as tecnologias de telefonia celular, comunicação via satélite e redes locais sem fio (padrão IEEE 802.11), associada ao crescimento e popularização de dispositivos computacionais móveis (PDAs - personal digital assistents - e notebooks) propiciou o surgimento de um novo paradigma computacional, denominado computação móvel.

A principal característica da computação móvel é o acesso a serviços e informações independentemente da localização do usuário e ainda que este esteja em movimento. De forma mais ampla, computação móvel é sinônimo de computação a qualquer momento e em qualquer lugar [1].

Sistemas de computação móvel são sistemas distribuídos interligados por uma rede local sem fio e que apresentam certo grau de transparência de mobilidade. Desta forma, a movimentação de recursos dentro de uma mesma rede ou entre redes distintas, inclusive usando tecnologias diferentes (por exemplo, mobilidade de uma rede IEEE 802.11 para uma rede Bluetooth) deve ser transparente aos usuários, sem afetar o funcionamento dos serviços e as computações em processamento. Algumas das características do ambiente de computação móvel são:

- O dinamismo da infra-estrutura de comunicação sem fio, com variações na largura de banda, altas taxas de erros de transmissão, desconexões freqüentes e insegurança na transmissão dos dados pelo ar;
- A limitação da fonte de alimentação dos dispositivos móveis (baterias) influencia o desenvolvimento de software móvel, pois este deve realizar computação de maneira eficiente, reduzindo ao máximo o consumo da fonte de energia;
- A alta heterogeneidade de dispositivos computacionais móveis, com características

distintas de processamento, memória, armazenamento estável, interface gráfica e forma de interação; e

- A mobilidade, como oportunidade para prover serviços baseados em contexto, no caso a localização.

Todas essas características tornam o desenvolvimento de software móvel uma tarefa árdua e contribuem para a importância de introduzir comportamento adaptativo. Em geral, a adaptação em dispositivos móveis é disparada por variações na qualidade da infra-estrutura de comunicação, o recurso mais limitado e imprevisível deste ambiente. Além disso, os dispositivos computacionais móveis apresentam geralmente limitação de recursos, tais como UCP, memória, armazenamento e bateria [15]. Desta maneira, modificações na disponibilidade desses recursos devem ensejar ações adaptativas na aplicação.

Considere o exemplo apresentado em [16]: um sistema móvel é equipado com um dispositivo GPS e também tem acesso a uma rede de celular sem fio. Existem duas formas distintas pelas quais o sistema pode inferir sua localização atual: solicitando a informação ao dispositivo GPS ou requisitando a um servidor de localização a identidade da célula em que se encontra. Ambas as soluções têm vantagens e desvantagens em relação ao funcionamento do sistema e seus recursos. A solução do GPS aumenta o nível de precisão da informação, mas exige mais energia da bateria. Por outro lado, a localização usando células de comunicação é menos precisa, mas caso o enlace de rede esteja sendo utilizado para alguma operação, não haverá consumo adicional de bateria.

Outro exemplo está apresentado em [37]. Nele, é apresentado um sistema de controle de desastres composto de duas aplicações, um observador e um suporte. O observador é responsável por monitorar uma área afetada por algum desastre e enviar suas observações para o suporte através de uma rede sem fio. O suporte é responsável por coordenar as operações de resgate baseado nas informações recebidas. Uma implantação típica desse sistema compreende diversas instâncias de observadores e um suporte. Em função da natureza da operação, são comuns variações de largura de banda entre os observadores e o suporte. Diante disso, o observador mantém configurações alternativas para as várias condições de disponibilidade da rede. Estas configurações diferem na qualidade dos dados enviados ao suporte. Dentre as alternativas de tipos de dados, encontram-se: vídeo de alta ou baixa definição, imagens em alta ou baixa definição e descrição textual detalhada ou resumida. Diante do estado de qualidade da rede, o observador seleciona

qual configuração irá utilizar.

2.5 Conclusão

Este capítulo apresentou diversos conceitos fundamentais sobre software auto-adaptativo, incluindo sua definição e caracterização, taxonomias que permitem classificar um sistema adaptativo segundo diferentes critérios, abordagens para desenvolvimento sistemático de software auto-adaptativo e área de uso, com a apresentação de dois cenários dinâmicos e heterogêneos, grades computacionais oportunistas e ambientes de computação móvel.

3 Arcabouço Adapta

Este capítulo apresenta o arcabouço Adapta, descrevendo em detalhes os componentes que compõem o arcabouço. Uma apresentação do mesmo é descrita na Seção 3.1; a visão geral de sua arquitetura é ilustrada na Seção 3.2; as interfaces dos componentes e as respectivas linguagens de reconfiguração estão na Seção 3.3; e a Seção 3.4 destaca a implementação dos componentes do arcabouço, as estruturas de dados usadas e diagramas de classe que demonstram a composição de cada um dos componentes.

3.1 Conceituação

O Adapta é um arcabouço para desenvolvimento de aplicações adaptativas que tomou por base o trabalho apresentado em [7, 8]. No Adapta, as aplicações são construídas segundo o padrão arquitetural *reflection* [6]. Desta forma, as aplicações se dividem em dois níveis: um meta-nível com informações sobre propriedades da aplicação, tornando-a autoconsciente de seus elementos e sua estrutura; e o nível-base responsável pela lógica da aplicação. As mudanças efetuadas no meta-nível afetam o comportamento dos objetos no nível base.

O Adapta compreende ainda um sistema de suporte à execução das aplicações adaptativas, responsável por monitorar o ambiente de execução da aplicação e notificar componentes e aplicações quando ocorrerem mudanças na disponibilidade de recursos que provoquem uma ação de reconfiguração.

Finalmente, o Adapta introduz uma linguagem de reconfiguração para definição dos elementos adaptáveis do modelo de dados de cada um dos componentes do arcabouço, um modelo de reconfiguração desses componentes, baseado em AOM - *Adaptive Object Model* [66], e suporte a dois mecanismos de reconfiguração, atualização dinâmica de parâmetros da aplicação e substituição dinâmica de algoritmos. Através dessa linguagem de reconfiguração, denominada *AdaptaML*, podem ser definidos os recursos computacionais a serem monitorados, os eventos a serem detectados e notificados, os elementos adaptáveis da aplicação (parâmetros, famílias de algoritmos e componentes) e as ações de

reconfiguração a serem aplicadas sobre o nível base.

3.2 Requisitos e Arquitetura

O desenvolvimento do Adapta foi dirigido pelos seguintes requisitos:

1. Prover uma infra-estrutura de monitoramento do ambiente computacional, com as seguintes características: flexível, permitindo incorporar novos monitores de recursos; disponível, permitindo instanciar monitores em tempo de execução sem interrupção do serviço; compartilhado por todas as aplicações em um mesmo nó computacional; que não sobrecarregue a rede com dados dos recursos sendo monitorados; que possa ser gerenciado pelo usuário do arcabouço ou administrador da rede usando operações como suspensão ou continuação do monitoramento de uma propriedade;

2. Prover um mecanismo de detecção e notificação de mudanças no ambiente computacional, com as seguintes características: flexível, permitindo definir novos eventos; disponível, permitindo instanciar novos eventos em tempo de execução sem interrupção do serviço; confiável na entrega de eventos a aplicações ou componentes registrados; que possa combinar o estado de diversos recursos na definição de eventos de interesse da aplicação; que possa definir eventos compostos, a partir de diversas fontes ao longo de toda a aplicação distribuída; que não sobrecarregue a rede com notificações de eventos desnecessárias;

3. Prover um mecanismo de adaptação, com as seguintes características: que ofereça suporte a métodos de reconfiguração genéricos, de forma a poder ser utilizado em uma grande variedade de aplicações; extensível para definir e incorporar novos mecanismos de reconfiguração, permitindo sua especialização de acordo com necessidades específicas da aplicação; que possa gerenciar as dependências entre os componentes de uma aplicação distribuída; que possa assegurar a manutenção da consistência e integridade de uma aplicação distribuída, através de um protocolo de sincronização de ações de reconfiguração entre componentes distribuídos interdependentes.

4. Prover uma linguagem de reconfiguração, com as seguintes características: que descreva o modelo de dados de cada aspecto de reconfiguração (monitoramento, detecção e notificação de eventos e adaptação dinâmica); que use elementos de alto-nível,

expressos em uma sintaxe XML; que possa ser facilmente editada pelo usuário do arcabouço para alterar certos aspectos de reconfiguração; e

5. Prover um arcabouço reconfigurável que possa modificar sua estrutura em tempo de execução, refletindo modificações efetuadas no modelo de dados de cada componente.

Considerando os requisitos especificados, definiu-se uma arquitetura para o arcabouço, composta pelos componentes ilustrados na Figura 3.1.

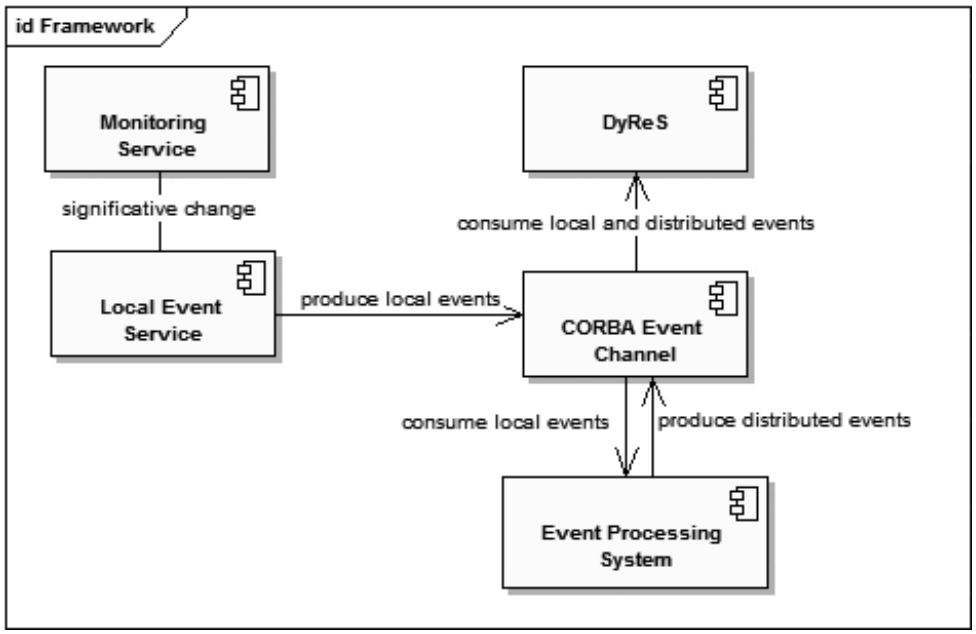

Figura 3.1: Componentes do Adapta

3.2.1 Serviço de Monitoramento

O monitoramento do ambiente distribuído desempenha um papel importante no processo de adaptação, oferecendo informações atualizadas relacionadas à disponibilidade de recursos computacionais em todo o ambiente [14]. Desta forma, ao perceber mudanças significativas no ambiente computacional, o mecanismo de adaptação pode executar políticas de reconfiguração, adequando o funcionamento da aplicação distribuída aos parâmetros de disponibilidade de recursos computacionais.

O Serviço de Monitoramento regularmente coleta dados do ambiente de execução e notifica quaisquer mudanças significativas na disponibilidade de recursos ao Serviço de Eventos Locais localizado no mesmo nó. Um Serviço de Monitoramento deve ser

instanciado em cada nó que possua recursos cujo monitoramento seja de interesse da aplicação ou componente adaptativo. O monitoramento compreende recursos de hardware (ex: UCP, memória, disco rígido ou a rede) e elementos de software (ex: quantidade de *threads* de uma aplicação, volume de troca de mensagens entre dois componentes). A infra-estrutura de monitoramento baseia-se em quatro conceitos principais:

- **Recursos**, que representam elementos de hardware e software, tais como: UCP, memória, interface de rede, disco rígido e aplicações.
- **Propriedades**, que representam atributos monitoráveis de recursos, tais como: percentual de uso da UCP, quantidade de memória disponível, largura de banda e latência de rede, espaço em disco disponível e total de threads de aplicação.
- **Monitores**, que monitoram, efetivamente, uma determinada propriedade em um ambiente computacional específico.
- **Faixas de operação**, que sinalizam mudanças significativas em uma determinada propriedade monitorada. Por exemplo, para monitorar o percentual de uso da UCP, poder-se-ia usar as seguintes faixas de operação: [0%, 20%), [20%, 40%), [40%, 75%), and [75%, 100%].

Em ambientes dinâmicos e heterogêneos, os requisitos de monitoramento da aplicação podem evoluir a qualquer momento e devem ainda atender a um conjunto mais amplo de plataformas computacionais com características específicas. Em razão disso, o Serviço de Monitoramento permite a adição, remoção e substituição dinâmica de monitores na infra-estrutura de monitoramento, sem que haja re-compilação de código ou interrupção de serviço.

3.2.2 Serviço de Eventos Locais

O Serviço de Eventos Locais é o componente do arcabouço responsável por detectar e notificar mudanças na disponibilidade de recursos no ambiente computacional, desacoplando o monitoramento do ambiente distribuído das ações de reconfiguração. Para isso, o Serviço de Eventos Locais recebe notificações das propriedades monitoradas pelo Serviço de Monitoramento localizado no mesmo nó. Esses dois componentes compreendem o sistema

de suporte a execução do Adapta, que executa em qualquer nó que possa hospedar uma aplicação adaptativa.

O desenvolvimento do Serviço de Eventos Locais foi baseado no modelo *publish-subscribe* [19], no qual um nó produtor publica eventos e nós consumidores registram-se no Serviço de Eventos Locais para recebê-los. As notificações de eventos devem ser produzidas e consumidas utilizando-se canais de eventos CORBA [42], pois é necessário notificar o Serviço de Eventos Distribuído do arcabouço que está localizado em outro nó na rede.

Eventos consistem em situações de disponibilidade de recursos que se estendem durante um determinado período de tempo chamado de **tempo de duração**. O tempo de duração é determinado na definição do evento e pode ser alterado dinamicamente pelo usuário do arcabouço. A avaliação de eventos baseia-se em expressões lógicas inseridas pelo usuário do arcabouço como parte da definição do evento. A fim de disparar a notificação de um evento, a expressão lógica correspondente deve permanecer válida durante todo o tempo de duração, evitando a geração de eventos quando situações temporárias ocorrem, por exemplo: picos de uso da UCP quando um aplicativo é aberto na máquina.

Aos eventos gerenciados pelo Serviço de Eventos Locais dá-se o nome de **eventos locais**. Estes eventos descrevem mudanças nos recursos computacionais obtidos em um nó único, como UCP, memória, disco rígido e interface de rede. **Eventos locais** iniciam o processo de adaptação de aplicações centralizadas e aplicações distribuídas cujas ações de reconfiguração independem dos demais nós da rede, e ainda provêem ciência das condições de disponibilidade de recursos computacionais no ambiente distribuído ao Serviço de Eventos Distribuído.

3.2.3 Serviço de Eventos Distribuído

Para certas aplicações distribuídas, a decisão de reconfiguração deve considerar a combinação de eventos detectados em nós distintos. Por exemplo, a migração de um componente de um nó para outro na rede deve levar em consideração a combinação do uso de UCP em todos os nós da rede. Para detectar e notificar a ocorrência desses **eventos compostos**, oriundos da combinação de dois ou mais **eventos locais**, o arcabouço dispõe do Serviço de Eventos Distribuído.

O Serviço de Eventos Distribuído do Adapta é o EPS (Event Processing System) [36]. A integração do EPS no arcabouço deu-se através da modificação de suas interfaces para o padrão CORBA-IDL, com uso de canais de eventos para produção e consumo de eventos. O EPS também foi refatorado para permitir a adição e remoção dinâmica de novos *eventos compostos*.

O EPS usa três parâmetros de processamento para lidar com alguns problemas típicos de ambientes distribuídos, como perda de mensagens, duplicação e troca de ordem, descritos a seguir:

- **Janela de detecção**, que indica um período de tempo dentro do qual uma instância de evento pode ser tratada. Uma instância de evento somente é válida se a diferença entre sua marca de tempo e o tempo atual for inferior a este valor. Este recurso evita o processamento de eventos muitos antigos que podem descrever uma condição do ambiente de execução que não ocorre mais;
- **Tempo de escalonamento**, que indica um tempo de espera para o processamento de uma determinada instância de evento recebida pelo sistema. Este mecanismo busca minimizar o impacto do atraso na entrega de eventos, aproximando a ordem de processamento dos eventos da ordem de envio dos mesmos; e
- **Tempo de concorrência**, que indica ao processador de eventos quando duas instâncias de um mesmo evento devem ser consideradas concorrentes (simultâneas) e tratadas como uma só.

3.2.4 Serviço de Reconfiguração Dinâmica

O Serviço de Reconfiguração Dinâmica (DyReS) compreende o mecanismo de adaptação que executa ações de reconfiguração nos componentes da aplicação em resposta a mudanças no ambiente de execução. Todas as aplicações ou componentes adaptativos são instanciados juntamente com um DyReS. Assim, em um mesmo nó podem co-existir diversas instâncias em execução do DyReS.

O DyReS inicia o processo de reconfiguração no momento de recebimento de um evento local ou composto. Para cada evento, o DyReS consulta uma tabela que associa o identificador único do evento com as ações de reconfiguração que serão realizadas

na aplicação ou componente. Atualmente, o DyReS é capaz de atualizar parâmetros dinamicamente e substituir algoritmos dinamicamente.

Na atualização de parâmetros, o usuário do arcabouço define um ou mais parâmetros atualizáveis. Cada um destes parâmetros contém um ou mais atributos. Por exemplo, a resolução de um vídeo consiste em dois atributos: altura e largura. Um parâmetro atualizável corresponde a um objeto do nível base da aplicação. Este objeto contém um método de *callback* que é invocado quando ocorrer um evento de interesse da aplicação, para efetivar atualização do parâmetro.

A substituição de algoritmos baseia-se no conceito de famílias de algoritmos que corresponde a um conjunto de dois ou mais objetos com uma mesma interface que podem ser substituídos em tempo de execução. Para tanto, utiliza-se um protocolo bem definido de transferência de estado. Um exemplo de uma família de algoritmos *hash* poderia conter implementações do MD5 e SHA-1.

Em uma aplicação distribuída, ações de reconfiguração tomadas em um componente podem afetar outros componentes da mesma aplicação. Por exemplo, um servidor de áudio pode modificar o algoritmo de codificação para consumir menos largura de banda da rede. Desta maneira, é necessário que os clientes da aplicação coordenem-se e substituam o algoritmo de decodificação. Para assegurar a consistência e integridade de todo o sistema distribuído, foi definido um protocolo de sincronização de ações de reconfiguração entre os componentes interdependentes.

O DyReS usa configuradores de componente [28] para gerenciamento de dependências entre componentes, com a definição de ganchos e clientes. Ganchos indicam os componentes nos quais se depende, enquanto clientes indicam os componentes dependentes. Logo, toda aplicação distribuída em execução no arcabouço, é constituída de cadeias de dependências formadas pelos configuradores. Eventos podem facilmente trafegar através dessa cadeia, transmitindo notificações de sincronização das ações de reconfiguração. Os configuradores de componente do Adapta são abertos e podem ser estendidos para adicionar novas ações adaptativas, como a substituição, replicação e migração de componentes ou outras técnicas de adaptação.

3.3 Interfaces e Linguagens de Reconfiguração

Um dos princípios que nortearam o desenvolvimento do Adapta foi a reconfigurabilidade de todo o arcabouço. Para isso, o modelo de dados de cada componente é descrito em **AdaptaML**, uma aplicação XML que compreende cada aspecto de reconfiguração: monitoramento, detecção e notificação de eventos locais e distribuídos e definição dos elementos adaptáveis da aplicação (parâmetros, famílias de algoritmos e componentes) e ações de reconfiguração. **AdaptaML** subdivide-se em quatro sublinguagens:

- ML (*Monitoring Language*), que define os recursos, propriedades, monitores e faixas de operação da infraestrutura de monitoramento;
- LEL (*Local Events Language*), que define os eventos locais;
- CEL (*Composite Events Language*), que define os eventos compostos e os parâmetros de processamento de eventos compostos; e
- RL (*Reconfiguration Language*), que define os elementos adaptáveis da aplicação e as ações de reconfiguração a serem tomadas em face de mudanças no ambiente computacional.

O modelo de dados de um componente escrito em **AdaptaML** é armazenado externamente à aplicação em um arquivo XML. Portanto, usuários do arcabouço podem modificar o arquivo paralelamente ao funcionamento dos componentes que compõem o arcabouço. O padrão AOM é usado para interpretar o modelo de dados e carregá-lo dinamicamente sem interrupção de serviço ou re-compilação de código. Isto é possível porque cada componente do Adapta possui em sua interface o método `loadObjectModel`. A realização de uma chamada a este método permite a alteração da estrutura do componente. Desta forma, o usuário do arcabouço pode adicionar, alterar ou remover recursos e propriedades monitoradas, eventos locais ou compostos ou modificar o meta-nível da aplicação e as ações de reconfiguração adotadas sobre o nível base.

A interpretação e construção dinâmica dos componentes usando AOM estão ilustradas na Figura 3.2. Nela, um usuário realiza a chamada ao método `loadObjectModel` na interface de qualquer um dos componentes do Adapta. Uma chamada a este método causa a chamada ao interpretador correspondente à sublinguagem do componente (representado pela interface `XMLParser`), que lê o arquivo de reconfiguração e produz uma

lista de descritores dos objetos que compõem o modelo de dados. Os descritores são submetidos a um processo de construção pelo objeto `Builder`, que decidirá se irá criar ou modificar o elemento adaptável (representado pela classe abstrata `Buildable`). Ao término do processo, todos os descritores presentes no sistema e que não foram retornados pelo interpretador serão removidos.

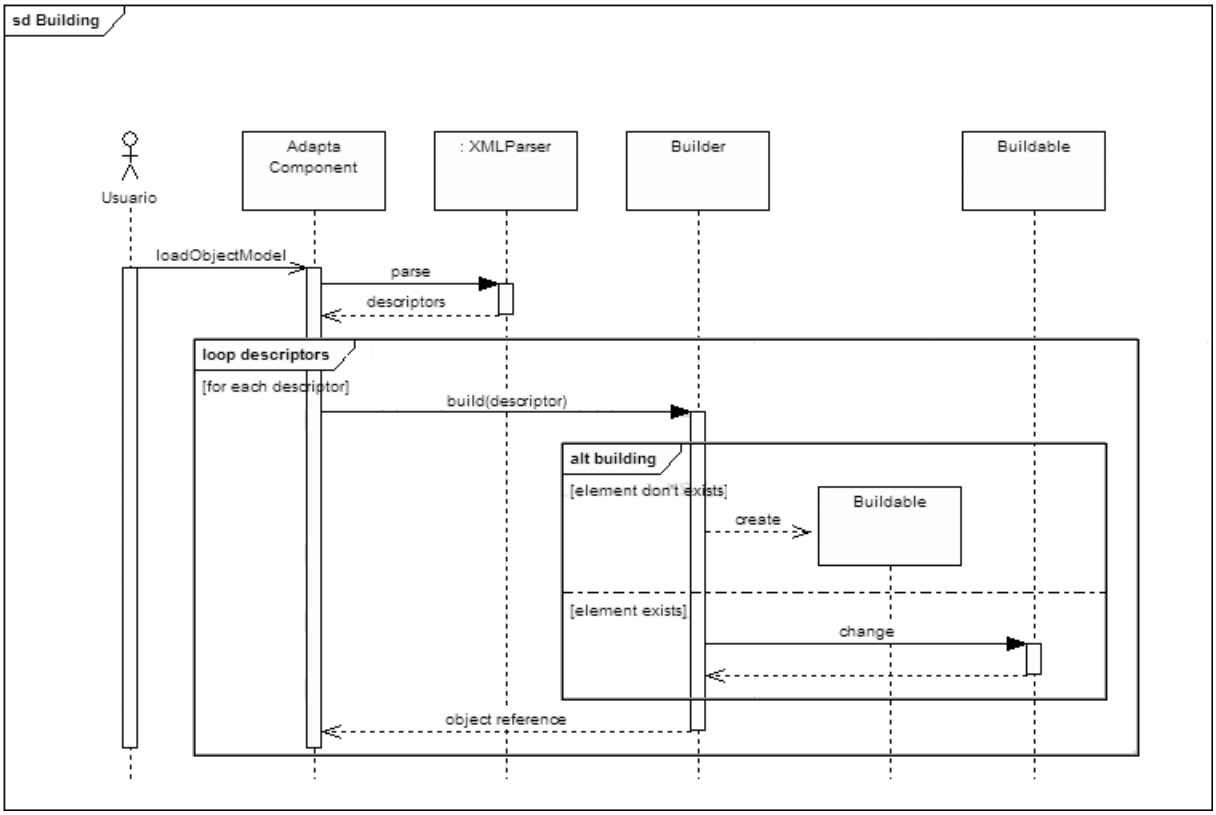

Figura 3.2: Interpretação e Construção Dinâmica

As seções a seguir apresentam em maiores detalhes a sublinguagem utilizada para a reconfiguração de cada componente do Adapta.

3.3.1 Serviço de Monitoramento

Usuários externos podem interagir com o Serviço de Monitoramento através de sua interface, ilustrada na Figura 3.3. O usuário do arcabouço pode: suspender ou continuar o monitoramento de uma propriedade, listar todos os recursos e propriedades do serviço, desativar todo o serviço e alterar o modelo de dados do serviço.

Ao ser realizada uma chamada ao método `loadObjectModel`, o componente realiza a leitura do arquivo de reconfiguração armazenado externamente e reflete as mudanças no modelo de objetos de monitoramento. Assim, adições, remoções ou alterações

```
1:  module adapta {
2:   module monitoring {
3:         module corba {
4:               typedef sequence<string> list;
5:
6:               exception InvalidResource { string resource; };
7:               exception InvalidProperty { string property; };
8:               exception InvalidFile {  string file; };
9:
10:              interface MonitoringService {
11:                    void loadObjectModel() raises (InvalidFile);
12:                    oneway void shutdown();
13:                    void suspend(in string rid, in string pid) raises (InvalidResource, InvalidProperty);
14:                    void resume(in string rid, in string pid) raises (InvalidResource, InvalidProperty);
15:                    string dump(in string rid, in string pid) raises (InvalidResource, InvalidProperty);
16:                    list resources();
17:                    list properties(in string rid) raises (InvalidResource);
18:              };
19:        };
20:  };
21: };
```

Figura 3.3: Interface do Serviço de Monitoramento

de recursos, propriedades, monitores ou faixas de monitoramento podem ser realizadas sem interrupção do serviço. Por exemplo, se um usuário desejar incluir um monitor de largura de banda de rede, ele deve modificar as definições do arquivo de reconfiguração do Serviço de Monitoramento e efetuar uma chamada a esse método.

Através da chamada ao método `suspend`, o componente interrompe o monitoramento de uma propriedade específica. Para continuar o monitoramento de uma propriedade suspensa, basta realizar uma chamada ao método `resume`. A desativação de todo o Serviço de Monitoramento, com a remoção das instâncias de recursos, propriedades e monitores é feita a partir de uma chamada ao método `shutdown`. Finalmente, o usuário pode obter informações acerca do monitoramento de uma determinada propriedade, a lista de recursos monitorados e a lista de propriedades monitoradas efetuando uma chamada aos métodos `dump`, `resources` e `properties`, respectivamente.

Linguagem de Reconfiguração do Serviço de Monitoramento: ML

A linguagem de reconfiguração do Serviço de Monitoramento é chamada de ML, descrita na figura 3.4. Os recursos monitorados são descritos usando `<resource>`. Cada recurso monitorado possui um identificador único (atributo **name**). As propriedades monitoradas são declaradas dentro de recursos utilizando-se a tag `<proprerty>`. De forma similar aos recursos, cada propriedade tem um identificador único (atributo **name**). Além disso, cada propriedade compreende ainda mais três elementos:

- `<frequency>`, que indica a frequência de monitoramento de uma propriedade em segundos;
- `<monitor>`, que indica a classe que implementa o monitoramento, totalmente qualificada, incluindo o nome dos pacotes (atributo **class**) e o caminho em que se localiza o código objeto da classe (atributo **path**);
- `<ranges>`, que descreve as faixas de operação usadas no monitoramento. Cada faixa é descrita com o elemento `<range>`, que consiste em limites inferior e superior (atributos **lowerbound** e **upperbound**, respectivamente).

```
1:  <monitoring>
2:   <resource name="">
3:        <property name="">
4:             <frequency value=""/>
5:             <monitor path="" class=""/>
6:             <ranges>
7:                  <range lowerbound="" upperbound=""/>
8:                  ...
9:                  <range lowerbound="" upperbound=""/>
10:            </ranges>
11:       </property>
12:
13:       <property ... />
14:  </resource>
15:
16:  <resource ... />
17: </monitoring>
```

Figura 3.4: ML

A ML pode ser usada para descrever adições, remoções ou alterações das propriedades de monitoramento. Para adicionar um monitor de percentual de uso da UCP, seriam feito os seguintes passos:

1. Criação de um `<resource>` com o nome UCP;
2. Criação de uma `<property>`, dentro de UCP, com o nome PercentualUso;
3. Definição do valor de `<frequency>` para 5, ou seja, a cada 5 segundos o monitor obtém os dados da UCP;
4. Definição dos atributos de `<monitor>`, o nome do monitor e o caminho em que se encontra o seu código objeto; e
5. Definição das `<ranges>`, como por exemplo: [0%, 30%[, [30%, 60%[, [60%, 75%[, [75%, 90%[e [90%, 100&[.

Para remover um monitor de um recurso basta comentar ou excluir o trecho do respectivo `<resource>`; já a remoção de apenas uma propriedade do recurso é feita no trecho respectivo `<property>`. Ao final de todas as edições na ML, o usuário do arcabouço deve invocar o método `loadObjectModel` para efetuar as mesmas em tempo de execução.

3.3.2 Serviço de Eventos Locais

A interface do Serviço de Eventos Locais está ilustrada na Figura 3.5. A notificação de um evento (`EventNotification`, nas linhas 9 a 13) compreende a identificação do evento, a marca de tempo e o nó (host) em que o evento foi produzido. A interface apresentada compreende operações de suspensão e ativação da detecção de um determinado evento, a desativação de todo o serviço, a listagem dos eventos ativos e suspensos e alteração do modelo de dados do componente.

```
1:  module adapt {
2:   module localevents {
3:         module corba {
4:               typedef sequence<string> list;
5:
6:               exception InvalidEvent { string event; };
7:               exception InvalidFile { string file; };
8:
9:               struct EventNotification {
10:                    string name;
11:                    string timeStamp;
12:                    string host;
13:              };
14:
15:              interface LocalEventService {
16:                    void loadObjectModel() raises (InvalidFile);
17:                    oneway void shutdown();
18:                    void suspend(in string eid) raises (InvalidEvent);
19:                    void resume(in string eid) raises (InvalidEvent);
20:                    string dump(in string eid) raises (InvalidEvent);
21:                    list activeEvents();
22:                    list suspendedEvents();
23:                    oneway void notifyMe(in string id, in double value);
24:              };
25:        };
26:  };
27: };
```

Figura 3.5: Interface do Serviço de Eventos Locais

Ao ser realizada uma chamada ao método `loadObjectModel`, o componente realiza a leitura do modelo de dados armazenado em um arquivo externo ao serviço. Esta chamada permite que eventos sejam adicionados, removidos ou alterados dinamicamente sem interrupção do serviço.

O usuário pode suspender ou re-ativar a detecção de um evento chamando, respectivamente, os métodos `suspend` e `resume`. A desativação de todo o componente é feita através de uma chamada ao método `shutdown`. Usuários podem recuperar informações relacionadas a um evento e listar os eventos ativos e suspensos invocando, respectivamente, os métodos `dump`, `activeEvents` e `suspendedEvents`.

A interface do Serviço de Eventos Locais compreende ainda o método *callback* `notifyMe`, usado pelo Serviço de Monitoramento para notificar mudanças na faixa de operação de determinado recurso computacional. O parâmetro **id** passado como argumento do método consiste na identificação inequívoca da propriedade monitorada, que é a concatenação entre o nome do recurso e da propriedade.

Linguagem de Reconfiguração do Serviço de Eventos Locais: LEL

A linguagem de reconfiguração do Serviço de Eventos Locais é chamada de LEL. As definições de eventos locais são feitas com o elemento `<event>`. Um `evento local` contém um identificador único (atributo **name**), uma expressão lógica (atributo **expression**) e o correspondente tempo de duração (atributo **duration-time**, expresso em segundos). A figura 3.6 ilustra a linguagem de reconfiguração LEL.

```
1: <local-events>
2:  <event name="" expression="" duration-time=""/>
3:  <event ... />
4: </local-events>
```

Figura 3.6: LEL

Expressões Lógicas de Eventos Locais

`Eventos locais` são descritos a partir de uma expressão lógica. Estas expressões combinam o identificador inequívoco composto pelo nome do recurso e da propriedade, operadores (descritos na tabela 3.1) e um valor. Desta forma, uma expressão lógica obedece o seguinte padrão:

```
<nome_recurso>:<nome_propriedade> <operador> <valor> ...
```

Por exemplo, a descrição de um `evento local` de sobrecarga em um nó, poderia ser: `UCP:PercentualDeUso grt 80 AND Memoria:EspacoLivre les 50`. Esta expressão lógica instrui o Serviço de Eventos Locais a disparar uma notificação quando o percentual de uso da UCP for maior que 80% e o espaço livro em memória for inferior a 50%.

Operador	Significado
geq	Operador de comparação $\geq$
leq	Operador de comparação $\leq$
grt	Operador de comparação $>$
les	Operador de comparação $<$
eql	Operador de comparação $=$
dif	Operador de comparação $\neq$
and	Operador lógico E
or	Operador lógico OU

Tabela 3.1: Operadores de Eventos Locais

Assim como na ML, a LEL pode ser usada para adicionar, removar ou alterar eventos locais. Para adicionar a detecção de um evento de redução na largura de banda, seria adicionado um novo `<event>` com os seguintes atributos: o nome ReducaoLarguraBanda, a expressão `Rede:LarguraBanda les 30`, onde Rede é um recurso e LarguraBanda uma de suas propriedades na ML e tempo de duração igual a 15, que indica que a expressão lógica deverá ser verdadeira durante 15 segundos para que o evento seja notificado às aplicações. De forma análoga, para remover uma definição de evento local, basta comentar ou excluir o trecho de código referente ao `event`. Ao final da edição da LEL, o usuário do arcabouço deve invocar o método `loadObjectModel`, para atualizar a lista de eventos locais detectados.

3.3.3 Serviço de Eventos Distribuído

A interface do Serviço de Eventos Distribuído está apresentada na Figura 3.7. Assim como os demais componentes do arcabouço, uma chamada ao método `loadObjectModel` permite que o modelo de dados do componente seja interpretado e carregado dinamicamente. Assim, o usuário do arcabouço pode adicionar, remover ou alterar `eventos compostos`

sem interrupção do serviço e ajustar os parâmetros de processamento, vistos na seção 3.2.3.

```
1:  module adapta {
2:   module distributedevents {
3:         module corba {
4:               exception InvalidFile { string filename; };
5:
6:               interface Eps {
7:                     void loadObjectModel(in string file) raises (InvalidFile);
8:                     oneway void shutdown();
9:                     void registerLes(in string ior);
10:              };
11:        };
12:  };
13: };
```

Figura 3.7: Interface do EPS

A desativação do Serviço de Eventos Distribuídos, com o encerramento do processamento de eventos, é realizado através de uma chamada ao método `shutdown`. O método `registerLes` é chamado pelos Serviços de Eventos Locais no momento da inicialização dos mesmos para registrarem seu canal de eventos junto ao Serviço de Eventos Distribuído.

Linguagem de Reconfiguração do Serviço de Eventos Distribuído: CEL

A linguagem de reconfiguração do Serviço de Eventos Distribuído é chamada de CEL e compreende a configuração dos parâmetros de processamento e a definição dos `eventos compostos`. Na Figura 3.8 está ilustrada a linguagem CEL.

```
1:  <eps>
2:   <log-file path=""/>
3:   <notification-file path=""/>
4:   <detection-window value=""/>
5:   <scheduling-time value=""/>
6:   <concurrence-time value=""/>
7:   <cleanup-time value=""/>
8:   <event name="" expression=""/>
9:   <event ... />
10:  </eps>
```

Figura 3.8: CEL

Os parâmetros de processamento descritos em CEL estão ilustrados abaixo:

- `<log-file>`, que indica onde será armazenado o arquivo de log do sistema. Se este

parâmetro for omitido, o log será exibido na tela do usuário;

- `<notification-file>`, que indica onde será armazenado o arquivo de notificações de entregas de eventos. Se este parâmetro for omitido, as notificações serão exibidas na tela do usuário;

- `<detection-window>`, que indica o período de tempo em que um evento pode ser tratado. Por exemplo: se o valor da janela de detecção for 10 segundos e a diferença entre a data de chegada do evento e a data de geração do evento for maior que 10 segundos, o evento é descartado e não é processado. Este recurso evita o processamento de eventos muitos antigos, que podem descrever uma condição do ambiente de execução que não ocorre mais;

- `<scheduling-time>`, que indica o período de tempo de espera para processamento de um evento. Por exemplo: se o valor do tempo de escalonamento for 5 segundos, o sistema irá reter os eventos recebidos durante este período, antes de processá-los. Este mecanismo busca minimizar o impacto do atraso na entrega de eventos, aproximando a ordem de processamento dos eventos da ordem de envio dos mesmos;

- `<concurrence-time>`, que indica o período de tempo no qual dois eventos são consideradas concorrentes. Por exemplo: se o tempo de concorrência for 5 segundos e a diferença da marca de tempo de dois eventos for inferior a esse valor, então os eventos são tratados simultaneamente como se fossem apenas um evento; e

- `<cleanup-time>`, que indica o período de tempo em que é feita uma coleta de lixo de todos os eventos que ultrapassaram o valor da janela de detecção.

Eventos compostos são descritos usando-se o elemento `<event>`, que compreende um identificador único (atributo **name**) e uma expressão lógica de avaliação (atributo **expression**).

Expressões Lógicas de Eventos Compostos

A definição da expressão lógica dos **eventos compostos** consiste na declaração do nome do **evento local** e o nó em que fora gerado relacionado a outros **eventos locais** através de dois operadores lógicos: o AND e o OR. A especificação do nó em que o **evento local** foi gerado pode ser feita usando-se um número IP válido, um nome no domínio de rede

ou a cláusula *localhost*. Desta forma, uma expressão lógica de **eventos compostos** segue o seguinte padrão:

```
[<no_geracao>]<nome_evento> <operador> ...
```

De forma similar às demais linguagens apresentadas até agora, a CEL permite a adição e remoção dinâmica de eventos compostos e ainda a alteração de parâmetros de processamento. Para adicionar um evento que compõe os dados de largura de banda de um determinado nó e o percentual de uso da UCP de outro nó, seria descrito um `<event>` com a seguinte expressão: `[ServidorLaboratorio]Alta_largura_banda AND [133.201.23.1]Baixo_consumo_UCP`. Esta expressão pode ser usada para iniciar o processo de migração de componentes do nó **ServidorLaboratorio** para o nó cujo IP é **133.201.23.1**, quando o consumo de UCP estiver baixo e a largura de banda observada no enlace daquele nó for suficiente para realizar a migração do código e dos dados. Para remover um evento composto basta comentar ou excluir o trecho de código referente ao `<event>`. Para ajustar a janela de detecção, por exemplo, aumentando o intervalo para processamento de eventos, basta modificar o atributo **value** do elemento `<detection-window>`. Ao final de todas as alterações em CEL, o usuário do arcabouço invoca o método `loadObjectModel` para efetuar as modificações em tempo de execução.

3.3.4 Serviço de Reconfiguração Dinâmica

Cada aplicação ou componente adaptativo é instanciado junto a um Serviço de Reconfiguração Dinâmica (DyReS). Assim como os demais componentes do arcabouço, a interface do DyReS compreende o método `loadObjectModel` que ao ser chamado, realiza a interpretação e construção dinâmica do modelo de dados do DyReS, que consiste no meta-nível da aplicação adaptativa (parâmetros, famílias de algoritmos e componentes) e estratégias de reconfiguração. A interface está ilustrada na figura 3.9.

Linguagem de Reconfiguração do DyReS: RL

A linguagem de reconfiguração do DyReS é chamada RL e está ilustrada na figura 3.10. RL é a sublinguagem mais complexa do **AdaptaML**. Através dela permite-se a definição de todos os elementos adaptáveis da aplicação (parâmetros, famílias de algoritmos e componentes), bem como a definição de estratégias de reconfiguração.

```
1:  module adapta {
2:   module dyres {
3:          module corba {
4:                 exception InvalidFile { string file; };
5:
6:                 interface DyReS {
7:                        void loadObjectModel() raises (InvalidFile);
8:                 };
9:          };
10:  };
11: };
```

Figura 3.9: Interface do DyReS

```
1:  <dynamic-reconfiguration>
2:
3:   <parameter name="" class="">
4:         <attribute name="" type=""/>
5:         <attribute ... />
6:   </parameter>
7:
8:   <parameter ... />
9:
10:  <object-family name="" proxy="">
11:        <state>
12:              <variable name=""/>
13:              <variable ... />
14:        </state>
15:
16:        <object name="" class=""/>
17:        <object ... />
18:  </object-family>
19:
20:  <object-family ... />
21:
22:  <component name="">
23:        <client name=""/>
24:        <client ... />
25:        <hook name=""/>
26:        <hook ... />
27:  </component>
28:
29:  <reconfiguration-strategy on-event="" notify="" notification="" timeout="">
30:        <action type="" target="" value=""/>
31:        <action ... />
32:  </reconfiguration-strategy>
33:
34:  <reconfiguration-strategy on-event="">
35:        <action type="" target="" value=""/>
36:        <action ... />
37:  </reconfiguration-strategy>
38:
39:  <reconfiguration-strategy ... />
40:
41: </dynamic-reconfiguration>
```

Figura 3.10: RL

Os parâmetros atualizáveis são descritos pelo elemento `<parameter>`. Todo parâmetro consiste em um identificador único (atributo **name**) e a classe no nível base

que o implementa (atributo **class**). O parâmetro atualizável contém um ou mais atributos, descritos pelo elemento `<attribute>`, com identificador único (atributo **name**) e um tipo de dados (atributo **type**). A descrição do tipo de dados do atributo é limitada pela própria natureza de XML, que dificulta o uso de tipos complexos, como coleções, listas ou objetos. Desta forma, Adapta permite apenas a declaração de tipos de dados primitivos, tais como strings, inteiros, pontos flutuantes, dentre outros. Esta limitação não é severa, já que parâmetros atualizáveis são na maioria dos casos numéricos como, por exemplo, a taxa de quadros enviados por segundo ou o índice de compressão usada.

As famílias de algoritmos são declaradas com o elemento `<object-family>`. Toda família possui um identificador único (atributo **name**), que corresponde à classe usada como proxy da família (atributo **proxy**). Os algoritmos que compõem a família são descritos pelo elemento `<object>`, com identificador único (atributo **name**) e a classe que implementa o algoritmo (atributo **class**). Durante o processo de substituição de algoritmos, pode ser necessária a transferência de estado entre o algoritmo sendo substituído e aquele que o substituirá. Para isso, o usuário do arcabouço deve descrever quais informações compõem o estado dos algoritmos da família através do elemento `<state>` e indicar as variáveis de estado (elemento `<variable`). Estas variáveis de estado consistem nos campos da classe que implementa o algoritmo. Por exemplo, o quadro sendo exibido de uma apresentação de vídeo é armazenado em uma variável na classe do servidor.

Os componentes (elemento `<component>`) descrevem a própria aplicação ou componente adaptativo pelo qual o DyReS é responsável. Assim, existe somente um elemento `<component>` dentro de cada arquivo RL e este pode conter zero ou mais ganchos (`<hook>`) e zero ou mais clientes (`<client>`). Os ganchos e os clientes descritos são registrados junto ao configurador de componentes, que resolve as dependências determinadas e efetuas as conexões com outros configuradores de componentes, montando uma cadeia de dependências de toda a aplicação distribuída.

O elemento `<reconfiguration-strategy>` descreve uma estratégia de reconfiguração, compreendendo uma ou mais ações (elemento `<action>`) que devem ser executadas no ato da notificação da ocorrência de um dado evento local ou composto (atributo **on-event**). Estas estratégias podem ser de dois tipos: sem sincronização, quando a reconfiguração é restrita ao componente em questão, sem envolver outros componentes da mesma aplicação distribuída; e com sincronização, quando a reconfiguração tomada em um componente afetará a consistência e a integridade de outros componentes da mesma

aplicação. Nesse último caso, a ação só terá início no momento em que todos os componentes dependentes tenham sido sincronizados ou caso o tempo limite de sincronização tenha se esgotado.

Para se definir uma estratégia de reconfiguração com sincronização, o elemento `<reconfiguration-strategy>` deve possuir os atributos opcionais **notify**, **notification** e **timeout**, que descrevem quem notificar (usando as palavras-chave clients, hooks, all ou o nome de um componente específico), o que notificar (o evento que será notificado através de cadeia de dependências) e o intervalo de tempo de tolerância de sincronização (Linhas 29-32), respectivamente. A definição de estratégias de reconfiguração sem sincronização é feita omitindo-se os atributos opcionais (Linhas 34-37).

As ações de reconfiguração (elemento `<action>`) inseridas em estratégias de reconfiguração possuem os seguintes atributos:

- O tipo de ação (atributo **type**), que atualmente pode ser **parameter**, usado para atualização dinâmica de parâmetros ou **object**, usado para substituição dinâmica de algoritmos.
- O elemento alvo da reconfiguração (atributo **target**), que poderá indiciar qual parâmetro será atualizado ou qual família de objetos será modificada; e
- O valor a ser refletido sobre o elemento alvo (atributo **value**), que em uma atualização de parâmetro é o nome da *callback* e a lista de parâmetros e em uma substituição de objetos, é o nome do novo algoritmo.

Alguns exemplos de ações de reconfiguração são: na modificação do valor da resolução de um vídeo, que compreende altura e largura, o `type` da ação seria **parameter**, o `target` é o nome do parâmetro atualizável - **Resolução** - e o `value` é a chamada do método `callback` com os parâmetros passados entre parênteses - setResolution(640, 480); na substituição dinâmica do algoritmo hash utilizado, o `type` é definido como **object**, o `target` é o nome da família de objetos - **Hash** - e o `value` é o nome do novo objeto a ser utilizado - **MD5**.

O usuário do arcabouço pode usar a RL para adicionar, remover ou alterar parâmetros atualizáveis, famílias de objetos e estratégias de reconfiguração. Para acrescentar um novo parâmetro atualizável (taxa de quadros, por exemplo), o usuário deve

definir um <parameter>, indicando seu nome e a classe que o implementa e os atributos do parâmetro - <attribute> -, neste caso apenas o valor da taxa de quadros. Ou para acrescentar uma família de objetos de codificação de vídeo, o usuário precisaria:

1. Definir o nome da família de objetos;
2. Determinar se esta família possuirá ou não estado (inserção do elemento <state>);
3. Se possuir estado, determinar as variáveis de estado - <variable>; e
4. Informar todos os <object> que compõem a família.

Para remover um parâmetro ou uma família de objetos ou uma estratégia de reconfiguração, basta comentar ou remover o código respectivo no arquivo da linguagem RL. Ao término da edição, as alterações serão confirmadas com uma chamada ao método `loadObjectModel`.

3.4 Implementação

Nas seções seguintes serão analisados detalhes de implementação de cada um dos componentes do Adapta.

3.4.1 Serviço de Monitoramento

Em cada nó que possua recursos de interesse à aplicação adaptativa deve ser executada uma instância do Serviço de Monitoramento, representado por uma instância da classe `MonitoringServiceImpl` que implementa a interface CORBA do serviço. As demais classes que compõem o serviço de monitoramento estão ilustradas na Figura 3.11.

A classe `MonitoringServiceImpl` contém uma coleção de todos os recursos (implementados pela classe `Resource`) contidos no modelo de dados do Serviço de Monitoramento. Estes, por sua vez possuem uma outra coleção com todas as propriedades monitoradas (implementadas pela classe `Property`). Cada propriedade monitorada está associada a um monitor específico.

Os monitores estendem a classe abstrata `Monitor` que possui métodos básicos, como a suspensão do monitoramento e a continuação de um monitoramento interrom-

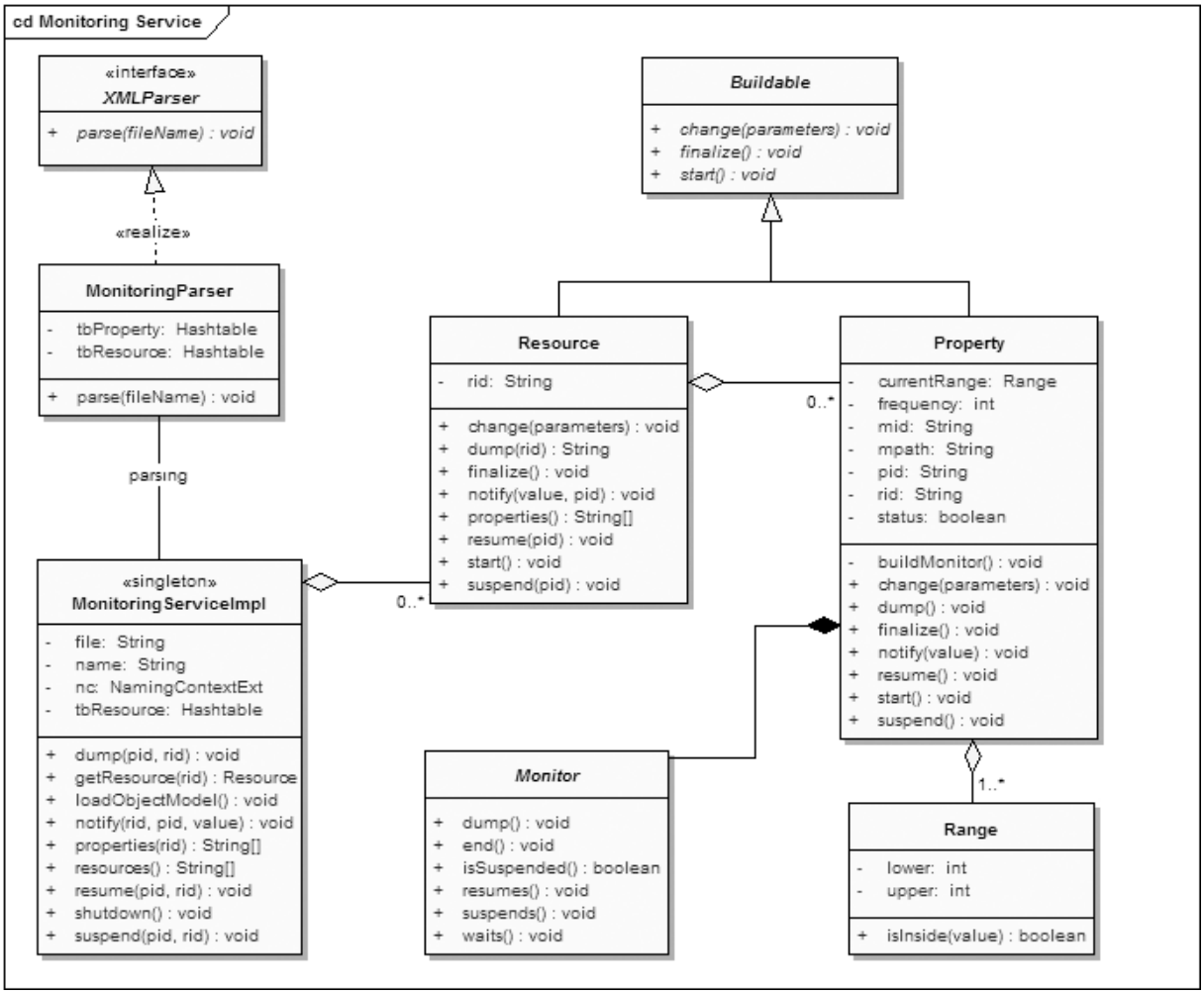

Figura 3.11: Serviço de Monitoramento

pido. O Adapta contém, dentre outros, monitores de percentual de uso de UCP, de uso de memória RAM e de espaço livre no disco rígido. Porém, usuários do arcabouço podem desenvolver novos monitores introduzindo novos requisitos não previstos ou substituir monitores para atender especificidades da plataforma monitorada. O processo de instanciação e substituição dinâmica de monitores é realizada pelo método `buildMonitor` (da classe `Property`), que usa os mecanismos da classe `URLClassLoader` para instanciar o código objeto do monitor em tempo de execução.

A interpretação do arquivo que contém o modelo de dados do Serviço de Monitoramento é realizada pela classe `MonitoringParser`. Esta classe verifica se houve inclusão, alteração ou remoção de recursos, propriedades, monitores e faixas de monitoramento, gerando os descritores que serão usados no processo de construção dinâmica do modelo de dados.

Além de coletar os valores das propriedades monitoradas do ambiente com-

putacional, o Serviço de Monitoramento notifica as mudanças significativas ao Serviço de Eventos Locais localizado no mesmo nó. Este processo está ilustrado na Figura 3.12. Ele é realizado periodicamente, sempre quando um objeto monitor coleta o valor da propriedade monitorada do ambiente computacional. Este monitor notifica à propriedade associada o valor obtido, através do método `notify` da classe `Property`. O objeto propriedade verifica se o valor está dentro da faixa de operação atual (campo `currentRange` da classe `Property`). Caso o valor esteja em uma faixa de operação diversa da atual, este objeto notifica o recurso que a contém através do método `notify` da classe `Resource`. Finalmente, o recurso notifica o Serviço de Monitoramento através do método `notify` da classe `MonitoringServiceImpl`. Ao término deste processo, a mudança significativa é enviada ao Serviço de Eventos Locais.

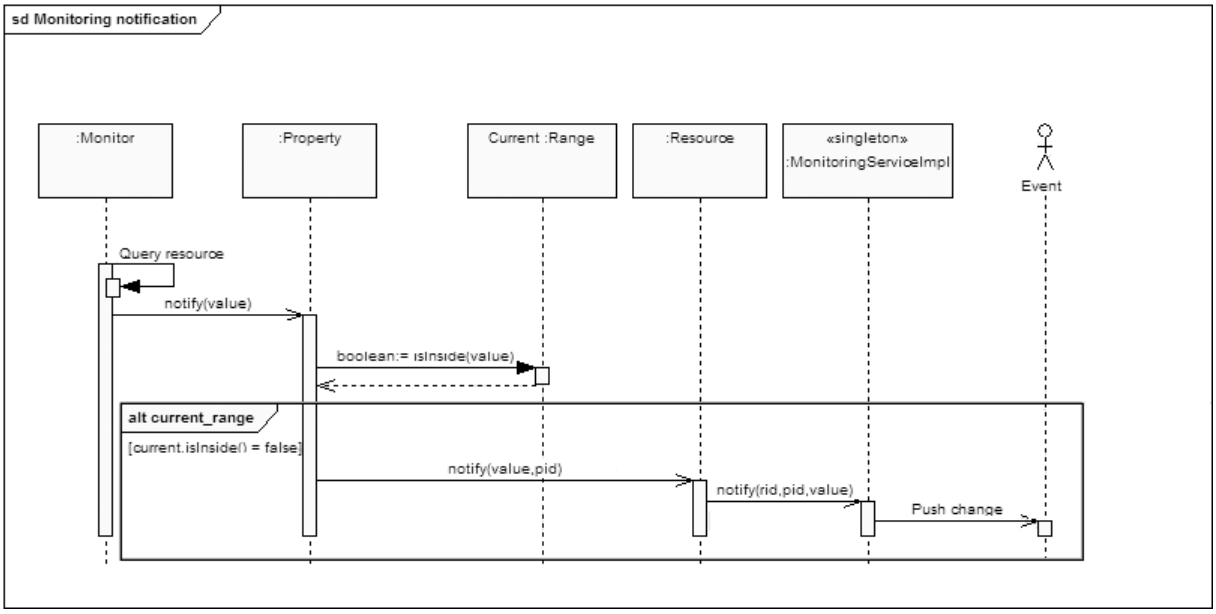

Figura 3.12: Notificação de Mudança Significativa

3.4.2 Serviço de Eventos Locais

O Serviço de Eventos Locais deve ser instanciado em todo nó onde é executado um Serviço de Monitoramento. A classe `LocalEventServiceImpl` implementa a interface CORBA do Serviço de Eventos Locais. Esta classe mantém todos os **eventos locais** registrados (classe `Event`). O diagrama das classes que compõem o Serviço de Eventos Locais encontra-se na Figura 3.13.

A classe `LocalEventServiceImpl` possui uma referência ao avaliador de expressões lógicas dos eventos locais (classe `Evaluator`). O avaliador contém a referência

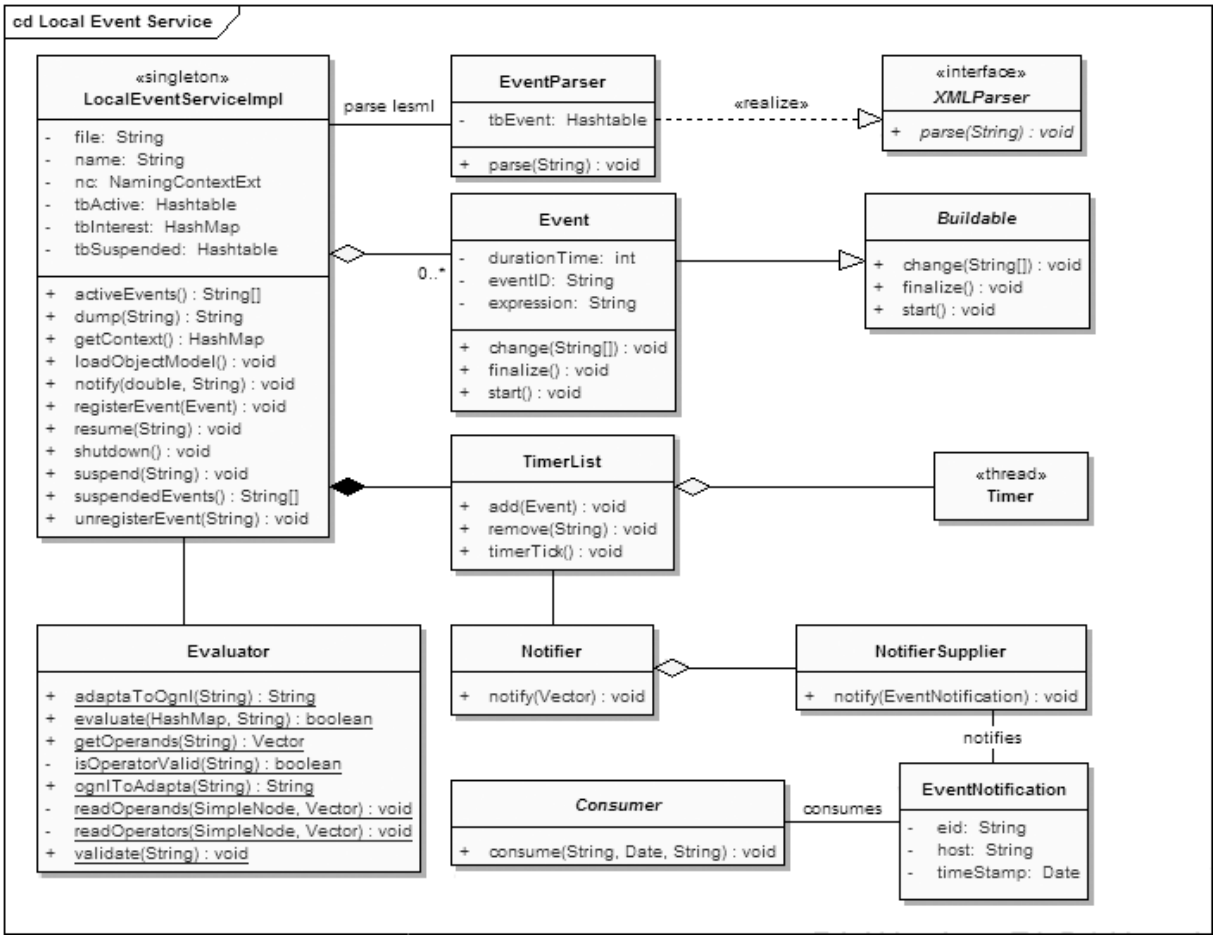

Figura 3.13: Serviço de Eventos Locais

a uma tabela, que representa o contexto computacional, com os valores atualizados das propriedades monitoradas. Sempre que há uma nova notificação de mudança significativa nas propriedades monitoradas, o avaliador analisa as expressões lógicas dos eventos registrados e detecta se alguma se tornou verdadeira.

A notificação de um evento detectado depende ainda de uma lista de temporização contida no Serviço de Eventos Locais. Nesta lista de temporização (implementada pela classe `TimerList`) ficam armazenados os eventos cuja expressão lógica fora avaliada como verdadeira, em ordem crescente pelo respectivo **tempo de duração**. A cada segundo, um temporizador (classe `Timer`) diminui os valores de tempo dos eventos na lista. Assim que um evento na lista atingir o valor zero ele é re-avaliado em relação aos novos valores das propriedades monitoradas no ambiente computacional.

Todos os eventos cuja expressão lógica for re-avaliada como verdadeira são enviados a um notificador (classe `Notifier`). Ao notificador cabe gerar uma mensagem contendo o nó (host) em que ocorreu o evento, a marca de tempo e o identificador do próprio

evento, gerando um novo objeto da classe `EventNotification`. A produção desta mensagem de evento no canal de eventos CORBA é realizada pela classe `NotifierSupplier`.

A detecção (com avaliação e re-avaliação da expressão lógica dos eventos) e a notificação de `eventos locais` está apresentada em maiores detalhes na Figura 3.14. Nela, pode-se ver que o processo é iniciado pelo Serviço de Monitoramento, que notifica uma mudança significativa em uma propriedade monitorada através do método `notifyMe`, da interface do `LocalEventServiceImpl`. Os parâmetros desse método são o valor da propriedade monitorada e a sua identificação, composta pelo nome do recurso e da propriedade (por exemplo, Rede:Latencia). Em seguida, o `Evaluator` avalia a expressão lógica de todos os `eventos locais` que contêm a propriedade monitorada. Todos os `eventos locais` que tiverem sido avaliados como verdadeiros são enviados ao `TimerList`, pelo tempo especificado no tempo de duração de cada evento.

Após expirar o tempo de duração de um evento na lista, o contexto dos novos valores das propriedades monitaradas é recuperado no `LocalEventServiceImpl` e ocorre uma re-avaliação deste `evento local`. Desta vez, se o evento for avaliado como verdadeiro, ele é submetido ao `Notifier`, que o envia ao `NotifierSupplier`, cuja responsabilidade é organizar uma estrutura de notificação de evento, com o nome do evento, a marca de tempo e o nó em que ele foi detectado. Finalmente, a notificação de ocorrência de um evento é enviada ao cliente através de canal de eventos CORBA.

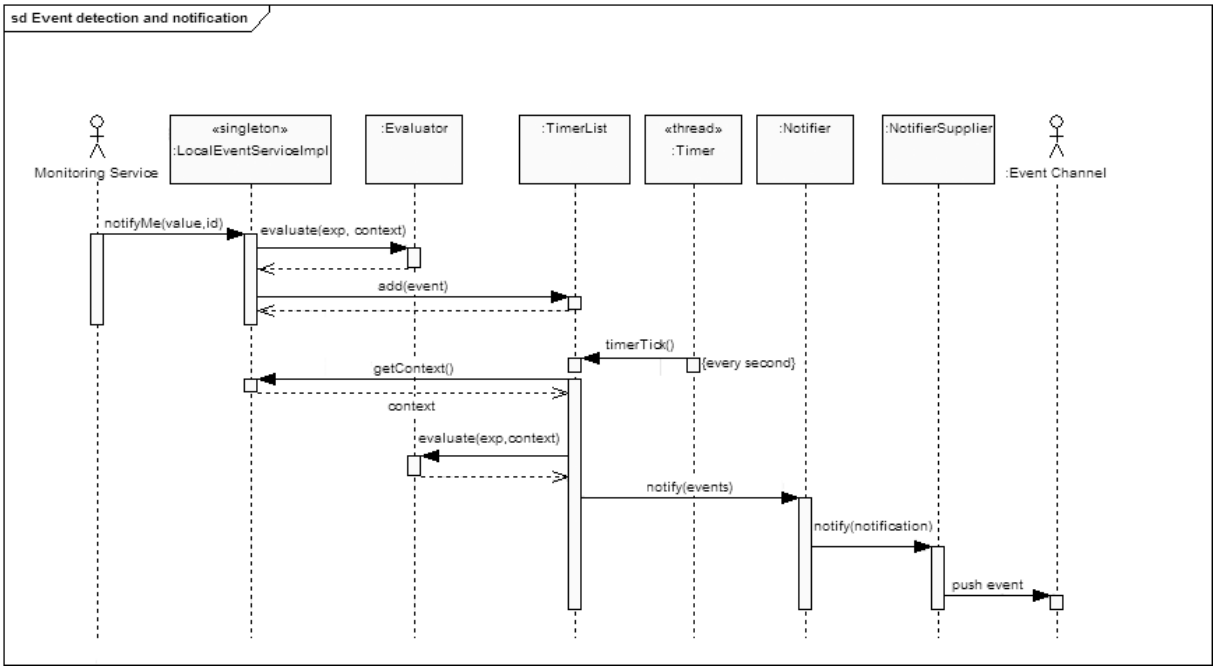

Figura 3.14: Detecção e Notificação de Eventos

3.4.3 Serviço de Eventos Distribuído

O Serviço de Eventos Distribuído, EPS, consiste em dois componentes principais: um gerenciador de interesse e um monitor de eventos. O gerenciador de interesse é usado para registrar ou cancelar o interesse de uma aplicação no recebimento de uma notificação de evento. O monitor de eventos consiste no processamento dos eventos e na notificação destes às aplicações registradas.

O EPS utiliza árvores para representação dos **eventos compostos**. Nestas árvores os **eventos locais** correspondem a folhas. Nós internos representam outros **eventos compostos** associados a um operador de composição específico (AND ou OR) e os nós raiz correspondem ao registro de interesse da aplicação em um determinado evento. Em alguns casos, **eventos compostos** distintos podem possuir sub-expressões comuns. Para evitar redundância de espaço e processamento nesses casos, o EPS compartilha sub-árvores comuns, criando as chamadas árvores agregadas.

O processamento dos eventos é de baixo para cima, inicia-se nas folhas da árvore até o nó raiz. Ao processador cabe encontrar a folha com o tipo de **evento local** correspondente e avisar aos nós pais respectivos a existência dessa nova instância. Ao ser avisado, o nó pai verifica a condição do **evento composto** e, caso seja verdadeira, notifica os respectivos nós pais. O processamento termina quando a condição verificada for falsa ou quando se atingir o nó raiz. Neste caso, a notificação do evento é enviada para as aplicações ou componentes registrados.

Uma árvore de processamento está exemplificada na Figura 3.15. Nela está ilustrado um registro de interesse no seguinte evento composto `(E1 or E2) and E3`. As folhas da árvore correspondem aos **eventos locais** E1, E2 e E3, os nós internos aos operadores de composição e o nó raiz à descrição do **evento composto**.

Imagine que o Serviço de Eventos Distribuído possua uma instância válida (ou seja, dentro da janela de detecção) do evento E1 e receba uma instância do evento E3. O processamento nesse caso dá-se da seguinte forma: o nó interno verifica a validade da expressão `E1 or E2`. Neste caso, ela é verdadeira pois o sistema possui uma instância válida de E1. Em seguida, o nó raiz verificará a validade da outra expressão, que é verdadeira. Nessa situação, uma notificação de evento composto é enviada às aplicações ou componentes registrados.

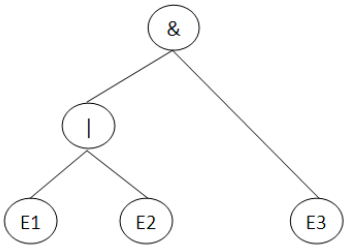

Figura 3.15: Árvore de Processamento

A fim de receber os **eventos locais** o Serviço de Eventos Distribuído coleta durante a sua inicialização as referências aos canais de evento de todos os Serviços Locais de Evento ao longo da rede. Isto é feito através de uma consulta ao serviço de nomes CORBA. Os Serviços de Eventos Locais que forem instanciados na rede posteriormente a inicialização do Serviço de Eventos Distribuído deverão registrar o seu canal de eventos através de uma chamada ao método `registerLes`, passando como parâmetro a IOR do canal.

3.4.4 Serviço de Reconfiguração Dinâmica

O Serviço de Reconfiguração Dinâmica (DyReS) consiste no mecanismo de adaptação responsável por aplicar ações de reconfiguração nos objetos do nível base da aplicação em resposta a mudanças no ambiente computacional. O DyReS abrange ainda a representação do meta-nível da aplicação, com a referência de todos os elementos adaptáveis desta, como parâmetros atualizáveis e famílias de algoritmos.

As classes que compõem o DyReS estão apresentadas no diagrama da Figura 3.16. A interface do DyReS é implementada pela classe `DyReSImpl`. Nela, compreendem-se todos os elementos adaptáveis da aplicação, tais como: parâmetros atualizáveis (classe `Parameter`), onde cada parâmetro possui um ou mais atributos, com nome, valor e tipo (classe `Attribute`); famílias de algoritmos (classe `ObjectFamily`); o componente junto o qual o DyReS é instanciado (classe `Component`) e; as estratégias de reconfiguração (classe `ReconfigurationStrategy`), compreendendo uma ou mais ações adaptativas (classe `Action`).

O mecanismo de reconfiguração dinâmica é mostrado na Figura 3.17. Uma thread do DyReS coleta eventos locais e compostos originados do canal de eventos do Serviço de Eventos Locais e do Serviço de Eventos Distribuído, respectivamente. Para

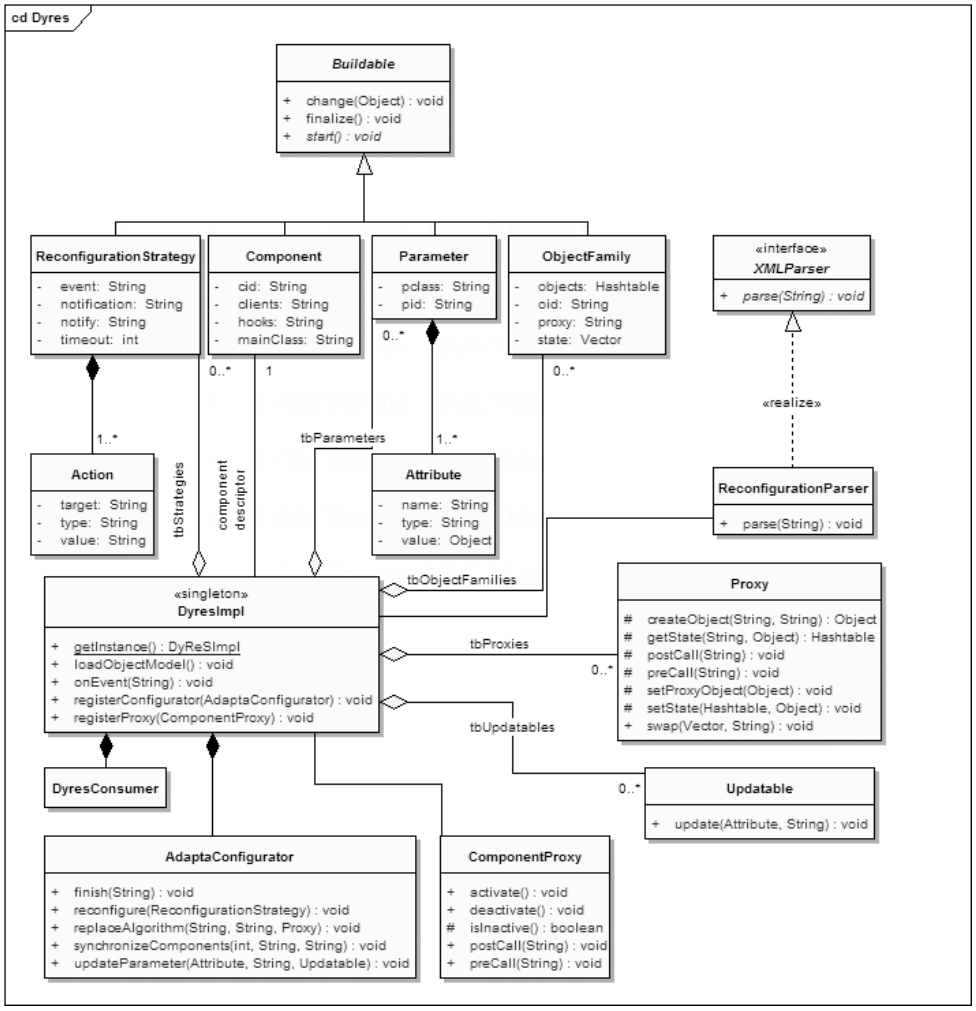

Figura 3.16: Diagrama de classes do DyReS

qualquer evento recebido, uma nova thread é instanciada para tratamento do evento através do método `onEvent` na classe `DyReSImpl`. Neste momento, `DyReSImpl` seleciona a estratégia de reconfiguração associada ao evento recebido.

Estratégias de reconfiguração contêm uma ou mais ações que serão executadas seqüencialmente na ordem em que forem definidas no arquivo de reconfiguração escrito pelo usuário do arcabouço. Antes do mecanismo de adaptação executar as ações de reconfiguração, o componente deve ser posto em um estágio de inatividade através de uma chamada ao método `deactivate` da classe `ComponentProxy`. A inatividade caracteriza-se pela impossibilidade de receber chamadas externas até que o processo de reconfiguração tenha sido concluído. As chamadas bloqueadas são enfileiradas por um proxy do componente, implementado pela classe `ComponentProxy` e tratadas ao término da reconfiguração.

Quaisquer ações de reconfiguração levam o componente a um estágio de inati-

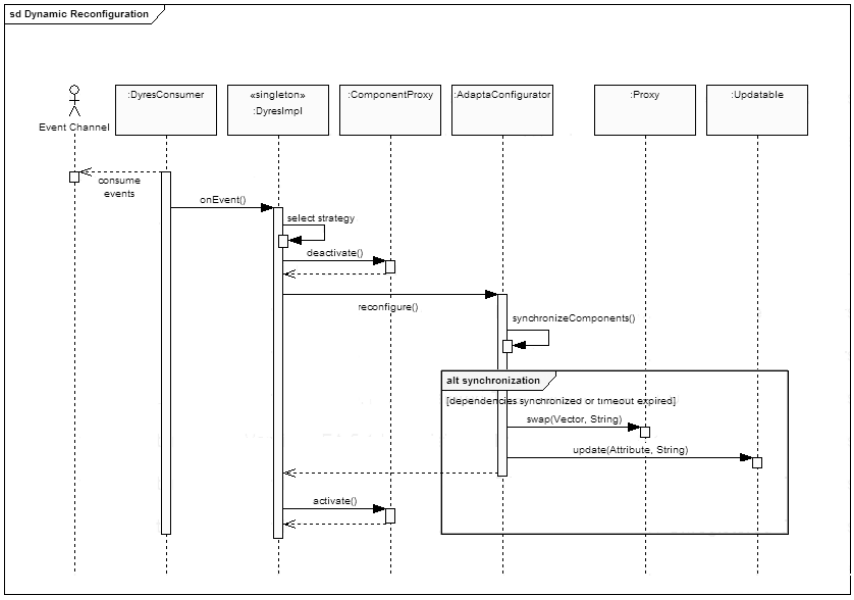

Figura 3.17: Reconfiguração Dinâmica no DyReS

vidade, de forma a manter a consistência de toda a aplicação em respeito ao momento de tomada da decisão de adaptação. Logo, até mesmo uma atualização de parâmetros interrompe a execução da aplicação para que as novas chamadas utilizem o valor do parâmetro já atualizado.

No momento em que o componente atingir a inatividade, `DyReSImpl` aciona o configurador de componentes (classe `AdaptaConfigurator`), através de uma chamada ao método `reconfigure`. O parâmetro deste método é a estratégia de reconfiguração associada ao evento recebido. Se houver necessidade, o configurador realizará a sincronização da reconfiguração com outros componentes interdependentes. A sincronização termina caso todos os componentes tenham sido sincronizados, ou caso o tempo limite de sincronização tenha se esgotado (por exemplo, na falha de um componente na rede).

Finalmente são realizadas as ações de substituição de algoritmos (método `swap` da classe `Proxy`) e de atualização de parâmetros (método `update` do parâmetro atualizável, classe `Updatable`). Concluída a reconfiguração, `DyReSImpl` reativa o componente através de uma chamada ao método `activate` de `ComponentProxy`, repassa a fila de chamadas bloqueadas e remove o proxy do componente.

Atualização Dinâmica de Parâmetros

A atualização de parâmetros usa uma abordagem baseada em *callbacks*. Assim, os desenvolvedores da aplicação adaptativa introduzem no código da classe que implementa o parâmetro um método *callback* que será chamada dinamicamente pelo configurador de componentes do Adapta.

Como exemplo, considere um mecanismo de compressão de dados com três métodos distintos, escolhidos de acordo com a capacidade de processamento da UCP. O primeiro minimiza o tempo de compressão, o segundo maximiza a taxa de compressão e o último é o método padrão. Estes métodos são representados respectivamente pelas constantes `BEST_SPEED`, `BEST_COMPRESSION` e `DEFAULT`. Caso o desenvolvedor do mecanismo de compressão de dados deseje converter o método de compressão em um parâmetro atualizável, ele deve inserir um método de *callback* (por exemplo, `setCompressionMethod`), cujo parâmetro seja uma das três constantes apresentadas.

A recuperação da referência do objeto que implementa o parâmetro atualizável e do método de *callback* que será chamado durante a atualização do valor do parâmetro é feita usando as informações constantes no arquivo de configuração escrito em RL pelo usuário do arcabouço.

Os objetos que contêm parâmetros atualizáveis estendem a classe `Updatable`. No momento de inicialização, os objetos dessa classe efetuam um registro junto à classe `DyReSImpl`. Desta maneira, o configurador de componentes do Adapta pode recuperar as instâncias que implementam o parâmetro a ser atualizado e efetuar uma chamada ao método *callback* obtido no arquivo de configuração.

Substituição Dinâmica de Algoritmos

A substituição entre os algoritmos pertencentes a uma mesma família usa o conceito de proxies para tornar transparente esse processo aos clientes. Os proxies (implementados pela classe `Proxy`) adicionam uma camada intermediária no fluxo de requisição e resposta. Conseqüentemente, os clientes comunicam-se apenas com o proxy e não com os membros da família de algoritmos. Desta maneira, no momento da decisão da substituição não há necessidade de religar as conexões entre os clientes e o objeto que implementa o novo algoritmo, pois toda esta lógica fica contida no proxy [6].

Os proxies das famílias de algoritmos são responsáveis por colocar em inatividade o objeto que contém o algoritmo em execução através do bloqueio e enfileiramento das novas chamadas recebidas. É também o proxy quem realiza a transferência do estado do objeto que implementa o algoritmo a ser substituído para o objeto contendo o algoritmo que o substituirá.

Mukhija e Glinz [40] apresenta dois algoritmos para a substituição de objetos, um chamado preguiçoso e outro guloso. No algoritmo preguiçoso, após a decisão de reconfiguração o mecanismo de adaptação aguarda até que o objeto em execução complete todo o processamento para então ser substituído. No algoritmo guloso, o objeto é imediatamente suspenso e o estado de processo salvo. Quando o novo objeto é posto no ar, ele continua o processamento no ponto em que o objeto anterior foi interrompido.

No arcabouço Adapta, usa-se o algoritmo preguiçoso ao invés do guloso. A justificativa para esta escolha é que o algoritmo preguiçoso é de implementação mais simples e ocasiona menor sobrecarga durante o processo de reconfiguração. Além disto, no algoritmo guloso em alguns casos o estado de execução no momento em que o objeto a ser substituído é suspenso pode não ser um estado válido para reiniciar o processamento no novo objeto. Por exemplo, em uma família de algoritmos de hash composta pelo MD-5 e o SHA-1 a interrupção no meio do processamento irá tornar inconsistente o cálculo do número hash, pois não existe nenhum estado válido em que se possa interromper um desses algoritmos e reiniciar o processamento em outro.

A substituição entre algoritmos pode ou não ensejar transferência de estado. Por exemplo, em um serviço de exibição de vídeos na Internet, o algoritmo usado na codificação do vídeo pode ser modificado dinamicamente de acordo com a disponibilidade de largura de banda. No momento da substituição do algoritmo, os meta-dados do vídeo (por exemplo: o idioma do aúdio e a legenda), bem como o último quadro exibido são transferidos dinamicamente para o novo algoritmo.

O proxy da família de objetos é o elemento responsável por realizar a transferência de estado entre os algoritmos usando uma política semelhante àquela vista em [61]. A recuperação das variáveis que compõem o estado da família de algoritmos é feita pelo configurador de componentes do Adapta a partir da leitura do conteúdo do arquivo de configuração escrito em RL e passada ao proxy no momento da substituição do algoritmo. Ao proxy cabe obter os valores de estado das variáveis do algoritmo em execução e gravar

estes valores no novo algoritmo.

Sincronização entre Componentes

A sincronização entre componentes interdependentes é realizada pelo configurador de componentes do Adapta, usando uma técnica de gerenciamento de dependências baseada no conceito de ganchos e clientes. Os ganchos consistem em todos os componentes dos quais um componente depende para seu funcionamento, enquanto os clientes indicam todos os componentes dependentes do funcionamento de um componente. Essa organização em ganchos e clientes forma uma cadeia de dependência entre os configuradores. Assim, eventos de sincronização podem trafegar através desta cadeia, transmitindo notificações das ações de reconfiguração sendo utilizadas.

A declaração dos ganchos e clientes é feita pelo usuário do arcabouço no arquivo de configuração escrito em RL. No momento de sua inicialização, o configurador de componentes do Adapta adquire, junto à classe `DyReSImpl`, os identificadores dos ganchos e clientes de seu componente. Estes identificadores são usados para resolver, no serviço de nomes de CORBA, a IOR de cada um. Ao término deste procedimento, o configurador de componentes armazena duas listas, uma com os ganchos e outra com os clientes de seu componente.

O processo de sincronização é prévio à reconfiguração e usado nas estratégias de reconfiguração em que forem informados os seguintes atributos: para quem notificar, clientes, ganchos ou ambos (usando, respectivamente, as palavras-chave `clients`, `hooks` ou `all`); o que notificar, ou seja, o nome do evento de sincronização que trafegará pela cadeia de dependência; e o tempo limite para o término da sincronização, que busca evitar que uma ação de sincronização se estenda indefinidamente em razão de falha no componente ou mesmo em toda a aplicação distribuída.

As ações de reconfiguração no componente prosseguirão assim que todos os componentes notificados, através dos ganchos e clientes, retornem indicando que concluíram a sincronização ou quando o tempo limite para sincronização houver expirado.

3.5 Conclusão

Este capítulo apresentou a arquitetura do arcabouço Adapta, seus componentes, interações e detalhes de implementação. Foi apresentada a AdaptaML, uma linguagem bem-estruturada para definição dos aspectos de reconfiguração: monitoramento do ambiente distribuído, detecção e notificação de mudanças na disponibilidade de recursos e reconfiguração dinâmica. O Adapta representa um avanço em relação ao trabalho apresentado em [7, 8], com as seguintes contribuições:

- Concepção de uma linguagem de reconfiguração, onde está descrito o modelo de dados de todos os aspectos da reconfiguração e de um mecanismo baseado em AOM que permite a interpretação e carga dinâmica do conteúdo expresso nesta linguagem no arcabouço, reconfigurando-o sem interrupção do serviço e mudança do código;
- Integração do EPS na arquitetura, com refatoramento de suas interfaces e re-escrita de código, para a definição de eventos compostos, disparados a partir da combinação de dois ou mais eventos;
- Desenvolvimento de dois mecanismos de adaptação, um para a atualização dinâmica dos parâmetros da aplicação e outro para a substituição dinâmica de algoritmos com um protocolo de transferência de estado bem-definido;
- Elaboração de um protocolo para sincronização de ações de reconfiguração em aplicações ou componentes com dependências no ambiente distribuído; e
- Aplicação e avaliação do arcabouço em novos estudos de caso conforme será apresentado no capítulo seguinte.

4 Avaliação do Arcabouço

Neste capítulo serão apresentados estudos de caso para avaliar a aplicabilidade e o funcionamento do arcabouço Adapta em algumas áreas da computação. Na Seção 4.1 será apresentada uma grade oportunista autônoma, com mecanismos de ciência de contexto, auto-configuração, auto-cura e auto-otimização. Enquanto isso, a Seção 4.2 ilustra o desenvolvimento de um servidor de vídeo adaptativo capaz de alterar dinamicamente a taxa de transmissão dos quadros do vídeo e introduzir dinamicamente um algoritmo de compressão do vídeo transmitido.

4.1 AutoGrid

O AutoGrid [55] é uma infra-estrutura de grade autônoma, que estende o middleware de grade Integrade [20] com habilidades de auto-gerenciamento. Isto é feito a partir do refatoramento de alguns dos componentes centrais do Integrade, adicionando a eles a capacidade de reconfiguração dinâmica.

O Integrade é um esforço multi-universitário para desenvolver uma infra-estrutura de grade com o objetivo de aproveitar o poder computacional ocioso das estações pessoais de trabalho para a execução de aplicações paralelas. A unidade arquitetural básica de uma grade Integrade é o aglomerado, uma coleção de máquinas normalmente conectadas por uma rede local. Aglomerados podem ser organizados em uma hierarquia, envolvendo um grande número de máquinas. Cada aglomerado possui um nó gerenciador que executa componentes do Integrade responsáveis por gerenciar os recursos computacionais do aglomerado e escalonar a execução de aplicações para os mesmos, gerenciar essa execução e realizar comunicações entre aglomerados. Os outros nós que compõem o Integrade são chamados estações de trabalho e exportam parte de seus recursos para os usuários da grade. Essas estações podem ser compartilhadas ou dedicadas.

O AutoGrid compreende, atualmente, quatro características autônomas: (a) ciência do contexto, que envolve o monitoramento de recursos da grade e a detecção de mudanças no ambiente computacional; (b) auto-configuração, que baseia-se em reflexão

computacional e consiste na representação da estrutura de componentes adaptativos que compõem o software da grade e o suporte a suas ações de reconfiguração; (c) auto-cura, que compreende um mecanismo flexível para tolerância a falhas de execução das aplicações; e (d) auto-otimização, através da substituição dinâmica do algoritmo de escalonamento de aplicações.

4.1.1 Ciência do Contexto

De forma a atingir um comportamento autônomo, é necessário que o AutoGrid tenha conhecimento do estado do ambiente de execução, o que inclui parâmetros individuais dos nós da grade, tais como: disponibilidade da UCP, memória utilizada, espaço em disco e parâmetros globais da grade, por exemplo: taxa de chegada de aplicações e o tempo médio entre falhas. O monitoramento do ambiente de execução da grade do AutoGrid consiste na execução do Serviço de Monitoramento em cada um dos nós da grade, com monitores que permitem, atualmente, coletar as informações de UCP, memória, disco rígido e taxa de falhas. Entretanto, outros monitores podem ser escritos e instanciados dinamicamente para introduzir outras propriedades de monitoramento.

Quando ocorrer uma mudança significativa na disponibilidade de um recurso, o Serviço de Monitoramento notifica o Serviço de Eventos Locais localizado em cada nó para avaliar a ocorrência de um evento. Assim que um evento é detectado, o Serviço de Eventos Locais notifica o Serviço de Eventos Distribuído, cuja função é detectar eventos compostos oriundos de dois ou mais nós da grade do AutoGrid. Todos os eventos locais e compostos detectados são notificados às instâncias do DyReS registradas.

4.1.2 Auto-Configuração

Cada componente adaptativo na arquitetura do AutoGrid incorpora uma instância do DyReS que corresponde ao seu meta-nível. É o DyReS o componente responsável por receber notificações de eventos locais e compostos e iniciar a reconfiguração em nome do componente associado. Atualmente, o AutoGrid permite duas ações de reconfiguração: mudança dinâmica de parâmetros das aplicações e substituição dinâmica de algoritmos com um protocolo bem-definido de transferência de estado.

Em geral, as reconfigurações em uma grade computacional precisam ser sincro-

nizadas entre componentes dependentes. Isto é muito comum em uma grade computacional em que vários componentes do middleware colaboram para execução das aplicações; neste caso, a alteração de um parâmetro ou mudança de um algoritmo pode originar a necessidade de reconfiguração em outros componentes. O gerenciamento de dependências baseia-se no uso de ganchos e clientes. Cada instância do DyReS mantém referências a todas as instâncias das quais ele depende (ganchos) e a todas as demais que dependem dele (clientes).

4.1.3 Auto-Cura

Tolerância a falhas consiste em um importante requisito para middleware de grade, pois a natureza dinâmica da infra-estrutura de grade, sua alta escalabilidade, grande heterogeneidade e a execução de tarefas longas aumentam a probabilidade de ocorrência de erros. As falhas têm diversas origens: falhas de hardware, como a queda de um nó da grade ou um particionamento na rede; erros de software, como exceções numéricas ou vazamento de memória; e outros motivos, como um usuário reiniciar um nó da grade ou a ocorrência sobrecarga de UCP.

A fim de prover as funcionalidades necessárias de tolerância a falhas para grades computacionais, diversos serviços devem estar disponíveis, tais como:

Detecção de falhas: nós da grade e aplicações devem ser constantemente monitorados por um serviço de detecção de falhas. Duas abordagens podem ser utilizadas: no modelo *push*, componentes da grade periodicamente enviam mensagens para um detector de falhas, anunciando que estão vivos. O monitor suspeita da ocorrência de falhas em um nó após um determinado intervalo de tempo sem recebimento de mensagens. Por outro lado, no modelo *pull* o detector de falhas envia periodicamente requisições (com a mensagem *está vivo?*) aos componentes da grade.

Gerenciamento de falhas na aplicação: diversas técnicas de recuperação de falhas podem ser aplicadas em grades computacionais de forma a continuar a execução de aplicações:

- *Reinício*: quando a execução de uma aplicação falha, ela é reiniciada do começo. Esta é a técnica de recuperação mais simples e sua principal desvantagem é a perda do tempo de computação em caso de falhas;

- *Replicação*: a mesma aplicação é submetida para execução um número de vezes, gerando várias réplicas da aplicação. Todas as réplicas são ativas e executam o mesmo código, com os mesmos parâmetros de entrada em nós distintos. Esta técnica tolera a ocorrência de até $n-1$ falhas, onde n é o número total de réplicas [12]; e

- *Pontos de controle*: periodicamente salva o estado do processo em um armazém estável, durante o tempo de execução livre de falhas. Quando ocorrer uma falha, o processo re-inicia a partir de um dos estados salvos, reduzindo o tempo perdido de computação. Cada estado salvo é chamado de ponto de controle. O uso dessa técnica impõe uma sobrecarga no tempo de execução, causado pela salva periódica do estado de execução. Maiores detalhes do mecanismo de pontos de controle do AutoGrid em [9, 10].

Armazém estável: estados de execução que permitem a recuperação do estado antes da falha das aplicações precisam ser armazenados em um repositório de dados que pode sobreviver a eventuais falhas dos nós da grade. Um armazém estável pode ser implementado usando uma abordagem centralizada ou distribuída. Na primeira, ele executará em um nó da grade dedicado; as desvantagens são a falta de escalabilidade, a presença de um ponto único de falha e a criação de um gargalo. Na segunda abordagem, os dados de recuperação são armazenados ao longo dos nós da grade, isto resolve os problemas da abordagem centralizada; porém, como os nós da grade são vulneráveis, os dados de recuperação podem ser perdidos.

Mecanismo de tolerância a falhas flexível

Grades computacionais são heterogêneas com relação às tarefas executadas (por exemplo, tarefas de longa duração, transações, tarefas em tempo real, etc) e com relação ao seu ambiente de execução (por exemplo, ambientes de execução altamente confiáveis e ambientes de execução não confiáveis). Esta heterogeneidade leva à necessidade de um mecanismo de tolerância a falhas flexível, com suporte a múltiplas técnicas de recuperação e capacidade de seleção da técnica mais apropriada com base nas características da tarefa e da confiabilidade do ambiente de execução.

Hwang et all [24] apresenta uma análise comparativa, baseada em simulações, de quatro técnicas de recuperação: reinício, replicação, pontos de controle e replicação

com pontos de controle. Os resultados apresentados demonstram que selecionar a técnica de recuperação de acordo com o ambiente de execução auxilia na diminuição do tempo aproximado de conclusão da tarefa. Por exemplo: se a taxa de falhas do ambiente for alta, então as técnicas que envolvem pontos de controle são melhores. Por outro lado, se a taxa de falhas do ambiente for baixa, o uso de pontos de controle não é apropriado por causa do *overhead*.

Considerando o que foi apresentado, um dos requisitos do AutoGrid é prover um mecanismo flexível de tolerância a falhas, que possa, inclusive, selecionar dinamicamente a quantidade de réplicas a serem geradas para a submissão de uma aplicação (no caso de a a técnica de recuperação utilizada envolver replicação).

Implementação

A implementação atual do AutoGrid utiliza as seguintes técnicas de recuperação de falhas na execução das aplicações: reinício, replicação e pontos de controle. Pode-se ainda combinar estas técnicas obtendo-se, por exemplo, replicação com pontos de controle. Atualmente, a escolha da técnica de recuperação a ser empregada no caso de falha de uma aplicação é realizada manualmente pelo usuário no momento de submissão da mesma. Caso o usuário escolha a técnica de replicação (ou uma combinação que a envolva), o AutoGrid decide, de forma autônoma, a quantidade de réplicas a serem geradas. Futuramente, pretende-se tornar autônoma a escolha da técnica de recuperação a ser adotada em função de diversos parâmetros, como a taxa de falhas do ambiente (MTBF).

A escolha da quantidade de réplicas a serem geradas para uma determinada solicitação de execução é baseada no MTBF e no tempo médio de execução da aplicação, ambos extraídos do banco de dados de execuções das aplicações, mantido por um componente do AutoGrid denominado `Execution Manager` (EM). Este banco de dados contém informações sobre cada execução da aplicação, como: o tempo em que a aplicação foi submetida e concluída, o tempo em que um processo falhou e o tempo em que este foi recuperado, dentre outras.

O `Global Resource Manager` (GRM) é componente do AutoGrid que armazena a quantidade de réplicas a serem geradas para uma nova submissão da aplicação na grade. Caso o usuário, ao submeter uma nova aplicação na grade, selecione alguma técnica de recuperação que envolva replicação, o GRM interage com o `Application Replication`

`Manager` (ARM) solicitando a geração das réplicas na grade. Ao ARM compete submeter os dados de entrada da computação nas réplicas, gerenciar a execução das réplicas e verificar a ocorrência de falhas e, ao término da execução de uma das réplicas, obter o resultado da computação e encerrar o processamento das demais réplicas.

Regularmente, o Serviço de Monitoramento inspeciona o banco de dados de execuções das aplicações e calcula o valor do MTBF do ambiente. O monitoramento ocorre a cada 30 minutos (frequência de monitoramento). Não há faixas de monitoramento para o MTBF. Portanto, qualquer novo valor de MTBF obtido é considerado uma mudança significativa. No instante de monitoramento, o monitor de MTBF obtém todas as falhas com o respectivo tempo em que estas ocorreram e calcula a média entre os tempos de cada falha. Ao final, o monitor calcula o tempo médio entre as falhas do ambiente.

Ao ser calculado o valor do MTBF, o Serviço de Monitoramento notifica o Serviço de Eventos Locais que houve uma mudança significativa no valor do MTBF. A definição do evento de mudança do MTBF é `Falhas.MTBF dif 0`. Dessa maneira, sempre que o valor do MTBF for diferente de zero, é realizada a ação de atualização desse valor. O tempo de duração do evento é 0 segundos, ou seja, não é realizada nenhuma re-avaliação do valor do MTBF. Ao avaliar a expressão, o Serviço de Eventos Locais cria uma notificação de evento e anexa a essa o valor do MTBF obtido do ambiente computacional. Essa notificação é enviada ao DyReS instanciado junto ao GRM.

No momento de recebimento de um pedido de reconfiguração, o DyReS atualiza o valor do MTBF no GRM. A atualização da quantidade de réplicas a serem geradas é realizada no momento de submissão de uma nova aplicação no GRM. Nele, encontra-se um algoritmo para calcular dinamicamente o número de réplicas a serem geradas. O algoritmo de seleção da quantidade de réplicas a serem usadas baseia-se em simulações realizadas em tempo de execução no momento de submissão da aplicação. Para cada submissão, o algoritmo cria um vetor de falhas baseado na distribuição exponencial (a partir do MTBF) e inicia um laço para determinar se a aplicação, com o número de réplicas usadas, é concluída em um tempo mais próximo possível do tempo de execução médio, determinado por um valor de tolerância. Se este valor de tolerância for muito alto, poucas réplicas serão necessárias, mas o tempo aproximado de término das aplicações será muito acentuado. Por outro lado, se a tolerância for um valor baixo, mais réplicas serão necessárias, aumentando o consumo de recursos computacionais da grade. Atualmente, o valor da tolerância no AutoGrid foi fixado em 20%, portanto, pode-se dizer que é tolerável

que uma aplicação cujo tempo médio de execução seja 24 horas tenha o tempo de execução em 30 horas aproximadamente. Futuramente deve-se definir um modelo matemático para cálculo do número de réplicas necessárias, aproximando o valor apresentado na simulação.

Avaliação

Um conjunto de simulações foi realizado com o objetivo de medir os benefícios da variação dinâmica da quantidade de réplicas a serem geradas em cada submissão de aplicação. A métrica usada foi o tempo aproximado do término da aplicação, obtido assim que a primeira réplica da aplicação for concluída.

As simulações consistem na execução da aplicação com diversos valores de MTBF e um número fixo de réplicas. Em cada execução da aplicação é gerado um vetor de falhas baseado na distribuição exponencial (a partir do MTBF). Para cada réplica, são mantidas duas variáveis: o tempo restante para término da aplicação (inicializado com o tempo de execução médio da aplicação) e o tempo acumulado de execução da aplicação. Para cada ocorrência de falha no vetor, é extraída a sua respectiva marca de tempo, e uma réplica é escolhida, aleatoriamente, para sofrer a falha. A marca de tempo obtida é adicionada ao tempo acumulado de execução da aplicação para todas as réplicas, e subtraída do tempo restante para término da aplicação, exceto para a réplica que falhou, neste caso o tempo restante para término da aplicação é reiniciado com o tempo de execução médio da aplicação. Esse laço é encerrado assim que: uma das réplicas atingir o tempo restante para término igual a zero que indica o término bem sucedido da aplicação, ou todas as réplicas atingirem um tempo de execução acumulado dez vezes maior que o tempo de execução médio da aplicação. Neste último caso, assume-se que a aplicação não irá conseguir concluir o seu processamento.

Os resultados desse conjunto de simulações, com a variação do tempo de execução médio da aplicação na ausência de falhas em 18, 36 e 72 horas, estão exibidos, respectivamente, nas Figuras 4.1, 4.2 e 4.3.

Analisando os dados da simulação, pode-se concluir que:

- Na medida em que a taxa de falhas aumenta (baixos valores de MTBF), uma maior quantidade de réplicas é necessária para manter o tempo de execução da aplicação dentro de uma faixa aceitável (um valor de tolerância);

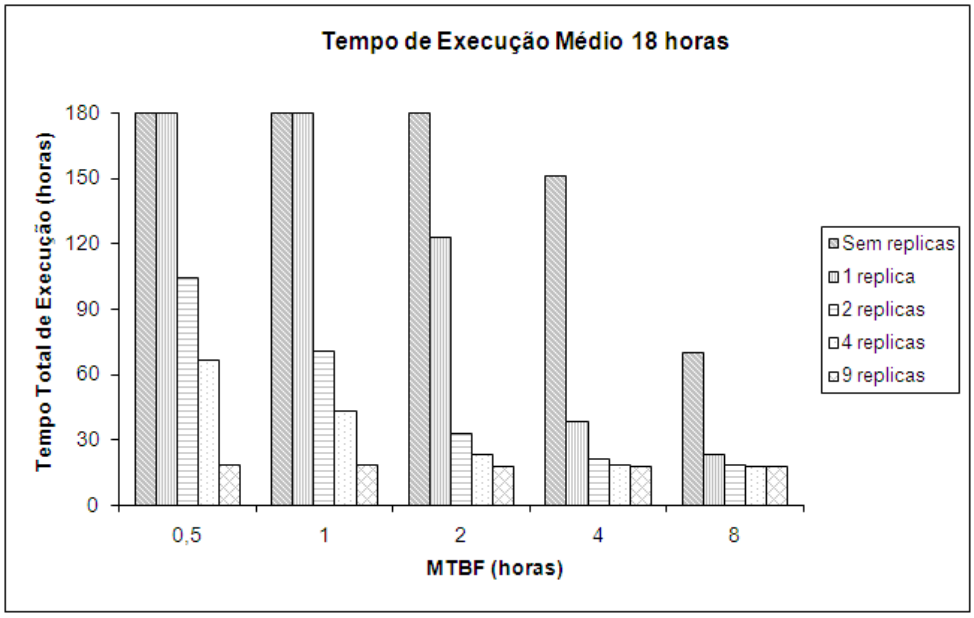

Figura 4.1: Tempo de Execução Média de 18 horas

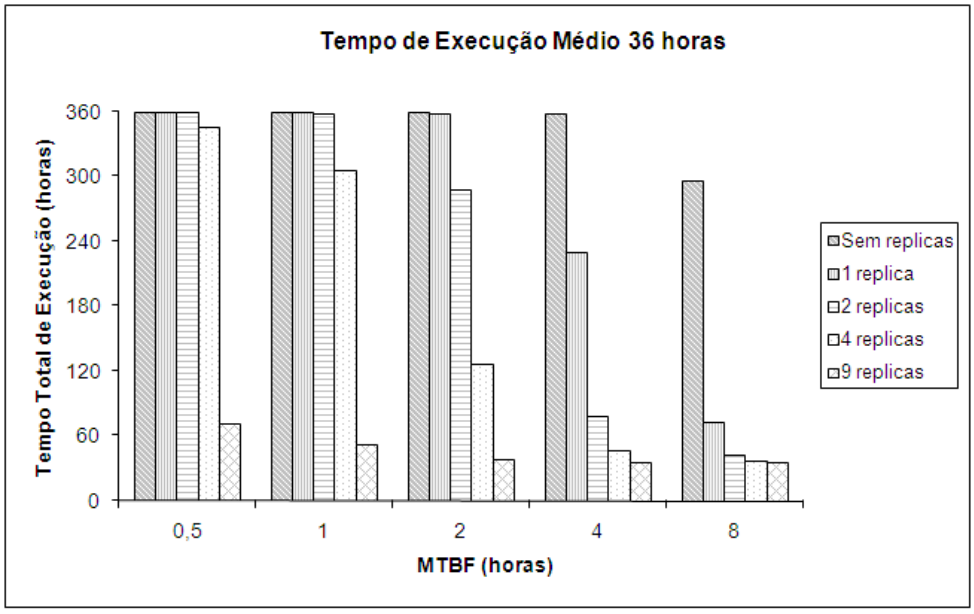

Figura 4.2: Tempo de Execução Média de 36 horas

- Se fixarmos o valor de MTBF do ambiente, não é vantajoso, em um determinado ponto, aumentar o número de réplicas, pois o ganho obtido seria mínimo ao mesmo tempo em que mais recursos da grade seriam consumidos. Por exemplo, na Figura 4.1, quando o MTBF tiver o valor de 4 ou 8, não haveria qualquer diferença significativa em usar 2, 4 ou 9 réplias. Na Figura 4.2, o mesmo poderia ser dito quando o MTBF for 8. Enquanto isso, na Figura 4.3, os ganhos seriam pequenos, porém significativos; e

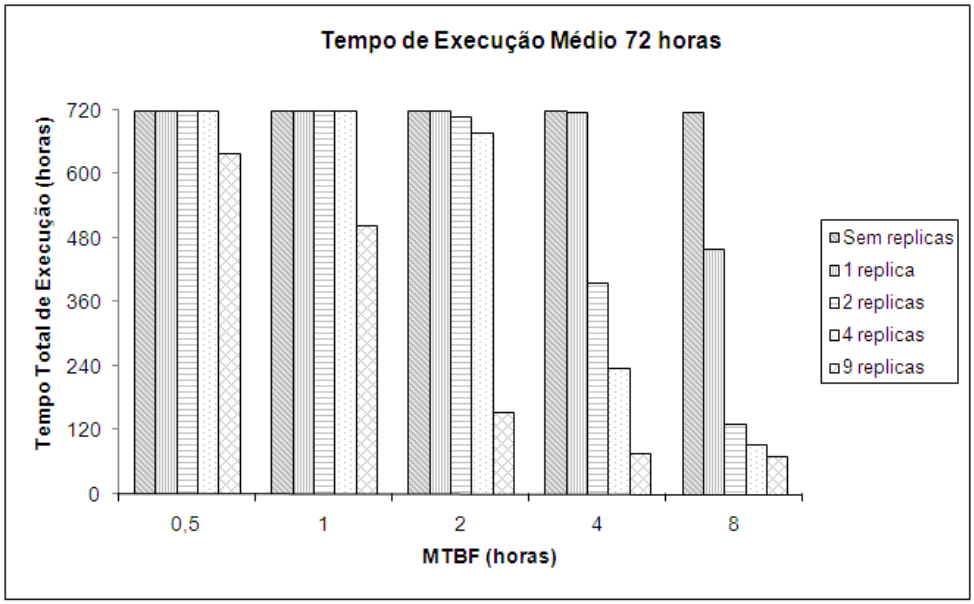

Figura 4.3: Tempo de Execução Média de 72 horas

- Algumas vezes grandes ganhos podem ser obtidos com uma pequena variação no número de réplicas. Por exemplo, na Figura 4.1, haveria uma enorme vantagem em usar 2 réplicas ao invés de 1 quando o valor do MTBF estiver no intervalo entre 0.5 e 2. Na Figura 4.2, o mesmo benefício seria obtido quando o MTBF estivesse contido no intervalo entre 2 e 4. Finalmente, a Figura 4.3 mostra o mesmo ganho, porém quando o MTBF estivesse em um intervalo posterior, de 4 a 8.

Considerando os dados apresentados na simulação, pode-se concluir que em ambientes com alta taxa de falhas, mais réplicas são necessárias para manter o tempo de execução da aplicação em uma faixa tolerável. Por outro lado, em ambientes com baixa taxa de falhas, pode-se dispor de uma menor quantidade de réplicas para isso, reduzindo o uso de recursos computacionais. Portanto, o AutoGrid se beneficia de um mecanismo de atualização dinâmica da quantidade de réplicas a serem geradas para uma submissão de aplicação, pois como o ambiente da grade computacional é altamente dinâmico e imprevisível, é esperado que a taxa de falhas do ambiente sofra alterações.

4.1.4 Auto-Otimização

Uma grade computacional compreende um ambiente altamente dinâmico. Este dinamismo pode ser observado mais claramente em grades computacionais oportunistas, que coletam o poder computacional ocioso de estações de trabalho pessoais para execução de aplicações

paralelas. Como as estações utilizadas não são dedicadas, a disponibilidade do poder computacional é variável. Por exemplo, o usuário da estação pode iniciar um programa que consuma quase toda a fatia de processamento que antes era usada em uma aplicação da grade. Isso tem impacto direto no desempenho das aplicações. A fim de evitar isso, alguns assuntos devem ser investigados, tais como: técnicas de balanceamento de carga, escalonamento adaptativo, re-escalonamento.

Dong et all [13] menciona o escalonamento adaptativo como a solução para o problema de escalonamento em que os algoritmos e parâmetros usados para tomar decisões de alocação de recursos computacionais podem ser alterados dinamicamente segundo as condições do ambiente de execução. Dong ainda afirma que o escalonamento adaptativo pode otimizar o desempenho geral da grade computacional e minimizar o tempo de resposta das aplicações.

Maheswaran et all [34] quantificaram o desempenho de algumas heurísticas de escalonamento que podem ser agrupadas em dois grandes grupos: heurísticas on-line e heurísticas batch. No modo on-line, uma tarefa (ou aplicação) é atribuída a um recurso na medida em que são submetidas à grade. No modo batch, as tarefas são agrupadas em um conjunto que é examinado para a atribuição aos recursos da grade em tempos pré-definidos chamados eventos de escalonamento. Cada uma dessas heurísticas compreende diversos algoritmos, tais como:

- MCT (`minimum completion time`), um algoritmo on-line que escalona a tarefa à máquina que resultar no menor tempo de término;
- MET (`minimum execution time`), um algoritmo on-line que escalona a tarefa à máquina que obtiver o menor tempo de execução, independentemente da fila de tarefas que contiver e do tempo de pronto;
- SA (`switching algorithm`), um algoritmo on-line que combina o MCT e o MET. Quando a grande estiver totalmente balanceada, o escalonador usa o MET; no momento que este causar desbalanceamento, é usado o MCT;
- Min-min, um algoritmo batch que organiza dois subconjuntos, um vetor com o tempo médio de execução de cada uma das tarefas e uma matriz que relaciona tarefas e nós computacionais com base no menor tempo de término. O algoritmo

seleciona a tarefa com o menor tempo de execução e a escalona ao nó computacional que a concluirá mais rapidamente;

- Max-min, um algoritmo batch similar ao min-min, exceto que ao invés de selecionar a tarefa com o menor tempo de execução, é selecionada a tarefa com o maior tempo de execução para ser atribuída ao nó computacional que a concluirá mais rapidamente;

- Sofrimento, um algoritmo batch no qual a idéia básica é determinar quanto cada tarefa seria prejudicada se não fosse escalonada no nó computacional que a executaria de forma mais eficiente. Em outras palavras, este algoritmo prioriza as tarefas de acordo com o valor que mede o prejuízo de cada uma. Esse valor de prejuízo é obtido como a diferença entre o melhor e o segundo melhor tempo de execução previsto para a tarefa.

Maheswaran usa simulações para demonstrar que existe sempre um algoritmo de escalonamento que é mais apropriado para uma determinada situação do ambiente de execução da grade. Por exemplo, algoritmos on-line são melhores quando a taxa de chegada de aplicações estiver acima de um valor pré-determinado da grade computacional. Nos demais casos, os algoritmos batch podem tomar melhores decisões de escalonamento porque organizam as aplicações em filas ou conjuntos parametrizados, por exemplo, com base no valor do prejuízo no algoritmo do Sofrimento. Dois objetivos de escalonamento são usados na análise, uma voltada a grade computacional (maximização do tempo de resposta) e outra voltada às tarefas em execução (minimização do tempo médio de espera).

Implementação

O GRM é o componente do AutoGrid que realiza o escalonamento das aplicações nos nós da grade. A implementação atual do AutoGrid permite a escolha do algoritmo de escalonamento a ser usado no ato da instanciação do GRM. Para tanto, utilizou-se o padrão Fábrica [6]. Os seguintes algoritmos estão disponíveis: o algoritmo padrão do Integrade, que escalona a tarefa para o primeiro nó computacional que satisfazer as restrições no momento de submissão da aplicação; e o OLB ou balanceamento de carga oportunista, que atribui a tarefa à primeira máquina que estiver ociosa, sem considerar o tempo de execução da aplicação. Caso mais de uma máquina esteja ociosa ao mesmo tempo, a escolha é arbitrária. Atualmente, está em desenvolvimento a implementação dos algoritmos

MCT, min-min e max-min.

A substituição dinâmica do algoritmo de escalonamento ainda não está implementada, mas o Adapta, através do DyReS, já provê os mecanismos básicos necessários para realizá-la. Para tanto, é necessário que sejam desenvolvidos dois monitores: um para a taxa de chegada de aplicações na grade, cujo valor é a quantidade de apicações submetidas na grade em um instante de tempo, e outro monitor para a média do percentual de uso da UCP nos recursos computacionais da grade. Os dados coletados por cada um desses monitores permitem que sejam tomadas decisões, tais como:

- Substituir um algoritmo batch por outro algoritmo on-line se a taxa de chegada de aplicações for superior a um valor pré-determinado;
- Substituir o algoritmo min-min pelo max-min, caso a média do percentual de uso da UCP nos recursos computacionais da grade diminua de acordo com um valor pré-estabelecido, pois o max-min pode maximizar a concorrência.

Um aspecto a ser considerado é o mecanismo de transferência de estado entre algoritmos de escalonamento. Considere, por exemplo, a substituição entre dois algoritmos batch. Nos algoritmos desta classe, filas são utilizadas para armazenar processos que aguardam a decisão de escalonamento. Essas filas são organizadas em alguns casos pelo tempo de chegada (por exemplo: max-min ou min-min) ou por um índice (por exemplo: o prejuízo no sofrimento). Na substituição entre dois algoritmos batch os processos constantes na fila do algoritmo antigo devem ser transferidos para a fila do novo algoritmo levando em consideração as particularidades de cada algoritmo. Em um outro exemplo, considere a substituição de um algoritmo batch por um algoritmo online. Nesse caso as aplicações que aguardavam em fila devem ser tratadas, no algoritmo online, ou como se estivessem sendo submetidas naquele determinado momento ou com a marca de tempo de submissão original da grade.

O processo de substituição dinâmica dos algoritmos de escalonamento na grade computacional compreenderá cinco estágios:

1. Ao receber um evento que ocasione a substituição do algoritmo de escalonamento da grade, um proxy é interposto entre o GRM e os demais componentes do AutoGrid;

2. Este proxy interceptará as submissões de aplicações e as armazenará em uma fila, até que o GRM atinja um estado de inatividade no momento em que o algoritmo em execução conclua o processo de escalonamento;

3. O estado do antigo algoritmo em execução é salvo e um novo algoritmo de escalonamento é instanciado;

4. O estado salvo é então carregado no novo algoritmo de escalonamento; e

5. O proxy é removido, a fila de submissões das aplicações é submetida ao GRM e todas as conexões entre os componentes da grade e o GRM são restabelecidas.

4.2 Servidor de *Stream* Adaptativo

Um servidor de `stream` é uma forma de consumir mídias (áudio e vídeo) ao redor da Internet que consiste no uso do arquivo de mídia enquanto ele está sendo recebido pelo cliente. Assim, em tempo real, os dados são transmitidos pelo servidor de `stream`, executados no tocador da máquina cliente e descartados logo em seguida ou armazenados em um *buffer* local. Este cache permite a manipulação do vídeo por parte do usuário através de operações como avançar, retroceder e interromper o vídeo, continuando-o em seguida.

Muitos servidores de `stream` usam o protocolo `Real Time Streaming Protocol` (RTSP) [58] para o controle da transferência dos dados de mídia com propriedades de tempo real. O protocolo RTSP torna possível a transferência sob demanda de dados de áudio e vídeo e permite estabelecer e controlar um ou mais fluxos de dados pertencentes a uma mesma apresentação. O protocolo define algumas operações básicas, dentre elas: `Setup`, que seleciona a mídia para transmissão e especifica como ela deve ser transportada; `Play`, que inicia a apresentação de uma mídia após operação bem sucedida de `Setup`; `Pause`, que interrompe temporariamente uma apresentação de mídia; e `Teardown`, que é usado para encerrar a sessão e interromper as apresentações.

A heterogeneidade das redes computacionais, abrangendo linhas discadas, comunicação banda larga e comunicação sem fio e o dinamismo dos meios de comunicação, em particular a Internet, torna necessário o uso de técnicas de adaptação dinâmica, de forma manter a qualidade de serviço do usuário, evitando problemas tais como a perda de quadros, a latência na exibição e o congelamento do vídeo. Por exemplo: um cliente

móvel ao atravessar um túnel pode, temporariamente, apresentar conectividade intermitente, causando perda de quadros. Uma solução seria diminuir a taxa de transferência dos quadros ou diminuir o tamanho dos mesmos para trafegar na rede.

O Adapta permite implementar diversas formas de comportamento adaptativo em um servidor de `stream` de vídeo, dentre elas: alteração dinâmica da resolução do vídeo, que pode ser realizada em tempo de execução no momento do envio do quadro ao cliente ou a partir da seleção de um mesmo vídeo em diferentes resoluções (fidelidades); alteração da profundidade de cores, inclusive com modificação do vídeo para tons de cinza ou preto e branco; alteração da taxa de envio dos quadros ao cliente; adição dinâmica de uma camada de compressão antes do envio do quadro que deve ser sincronizada com a adição de camada de descompressão no cliente. Em todos estes casos, as ações de reconfiguração são tomadas a partir de dados obtidos da infra-estrutura de rede, tais como: latência, largura de banda, dentre outros.

4.2.1 Implementação

Um servidor de `stream` de vídeo, baseado em Java e no protocolo RTSP, desenvolvido pela `Universidade de Maryland` foi usado como base para desenvolvimento dos mecanismos adaptativos. Esse servidor de `stream` extrai, periodicamente, quadros de um vídeo em formato MJpeg e os envia ao cliente. Os quadros são enviados em uma thread usando UDP e não há armazenamento de quadros em um buffer na máquina cliente. O Adapta foi usado para incluir duas ações de reconfiguração no servidor de `stream`: atualização dinâmica da taxa de quadros e adição de uma camada de compressão no servidor com sincronização junto aos clientes.

Nesse software, um servidor (classe `Server`) envia, em intervalos regulares, quadros a um cliente representado por uma interface gráfica (classe `Client`). O Adapta, dinamicamente, pode alterar a taxa de envio de quadros e pode inserir ou remover um algoritmo de compressão antes do envio do quadro. Em ambos os casos, é necessário o desenvolvimento de monitores de rede, por exemplo: monitor de largura de banda disponível e monitor de latência. Diante da complexidade para se desenvolver esse monitores, foi apenas implementada a camada de reconfiguração do servidor.

No servidor existe um atributo `frameRate`, determinado em milissegundos, que indica a periodicidade no envio de quadros. Normalmente, o servidor enviará 10 quadros

por segundo, ou seja, 600 quadros por minuto. Para aumentar ou reduzir esse valor o DyReS modifica o atributo `frameRate` da classe `Server`.

Para adicionar dinamicamente uma camada de compressão, no servidor foram definidos dois algoritmos de envio de quadros, um que faz uso de compressão e outro que não utiliza esse recurso. Por padrão, o servidor usa o algoritmo sem compressão. Ao receber o evento de diminuição na largura de banda da rede, ocorrem os seguintes passos:

1. O DyReS do servidor verifica a necessidade de sincronização com o cliente e envia uma notificação de reconfiguração ao DyReS do cliente;
2. O DyReS do cliente requisita ao proxy do cliente que modifique o algoritmo de recebimento de quadros;
3. O proxy do cliente instancia o novo algoritmo de recebimento e redireciona as novas chamadas a ele;
4. O DyReS do cliente informa ao DyReS do servidor o término da mudança do algoritmo de recebimento de quadros;
5. O DyReS do servidor requisita ao proxy do servidor que modifique o algoritmo de envio de quadros; e
6. O proxy do servidor instancia o novo algoritmo de envio e redireciona as novas chamadas a ele.

A alteração da taxa de quadros pode melhorar a qualidade de serviço para os clientes do servidor de vídeo. Normalmente, um servidor de vídeo envia 10 quadros por segundo, em um total de 600 quadros por minuto. Ao reduzir a quantidade de quadros enviados por segundo, diminui-se também a largura de banda usada pelo servidor, evitando problemas de apresentação em razão do congestionamento de quadros. Reduzindo ou restabelecendo a taxa de quadros, permite-se que o cliente do servidor assista o vídeo ininterruptamente em casos de queda na largura da banda ou aumento da latência da rede, melhorando a qualidade do serviço.

Avaliação

Um experimento foi realizado com o objetivo de mensurar os benefícios da compressão de dados, em relação à economia em largura de banda. A métrica utilizada foi a quantidade

de quadros perdidos. O experimento consiste na execução do servidor de vídeo em três larguras de bandas distintas (256 Kbps, 512 Kbps e 1 Mbps).

O experimento consiste na transmissão de um vídeo com 500 quadros de uma máquina servidora para máquinas clientes. Como essa transmissão acontece usando uma conexão UDP, é natural que alguns quadros sejam perdidos em função da baixa largura de banda disponível para envio. Para manter a quantidade de quadros perdidos, os clientes mantêm uma variável incrementada assim que um novo quadro é recebido. Para assegurar que a largura de banda entre o servidor e os clientes seja exatamente a definida na mecânica do experimento são usados limitadores de rede. Os resultados obtidos pelo experimento estão apresentados na Figura 4.4.

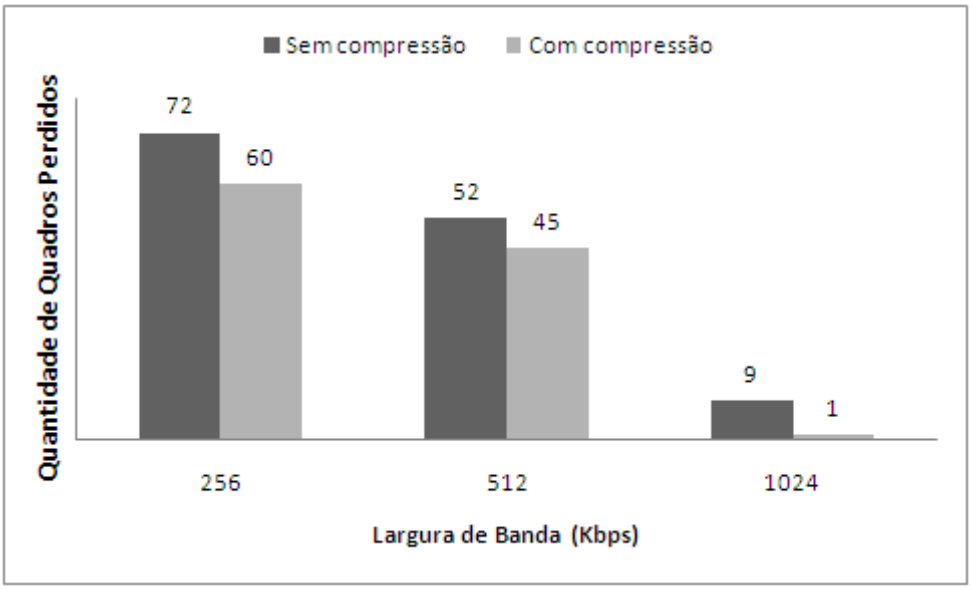

Figura 4.4: Quantidade de Quadros Perdidos

Pode-se concluir, com base nos resultados ilustrados, que o ganho obtido pela compressão de dados é relativamente pequeno: na largura de banda de 256 Kbps, 72 quadros são perdidos em média na execução do vídeo sem compressão. Usando compressão esse número caí para 60 quadros, um ganho de 16,67%; em 512 Kbps, o ganho é ainda mais reduzido. A quantidade de quadros perdidos sem compressão é de 52, contra 45 usando compressão de dados (ganho de 13,46%); e finalmente, usando 1 Mbps apenas 9 quadros são perdidos sem compressão, contra 1 quadro perdido usando compressão de dados.

Diante dos dados apresentados na simulação, verificou-se o baixo ganho no uso de compressão de dados antes do envio de quadros. Isso pode ser explicado porque o

formato MJpeg é um formato comprimido. Espera-se que outras técnicas possam obter melhores resultados como, por exemplo, a modificação da resolução do vídeo ou a alteração da profundida de cores. No entanto, observamos que o Adapta mostrou-se adequado para a incorporação adaptativa da camada de proteção, facilitando consideravelmente sua implementação.

4.2.2 Conclusão

Neste capítulo, foi apresentado o AutoGrid, uma grade computacional autônoma com mecanismos de ciência de contexto, auto-configuração, auto-cura e auto-otimização implementados usando o Adapta. O sistema de suporte a execução, juntamente com o DyReS, possibilitam a ciência do contexto e auto-configuração sem modificação do código do AutoGrid. A auto-cura foi desenvolvida através de um mecanismo de atualização dinâmica da quantidade de réplicas a serem geradas para cada submissão de aplicações na grade. Ainda no contexto do AutoGrid, discorreu-se acerca de mecanismos de auto-otimização, através de um mecanismo de substituição dinâmica do algoritmo de escalonamento de aplicações utilizado. Atualmente, é possível modificar estaticamente o algoritmo usado, alterando apenas o arquivo de inicialização do AutoGrid. Entretanto, com a introdução do Adapta, a seleção do algoritmo será realizada dinamicamente, sem interrupção do serviço. Ao desenvolvedor da grade, competirá somente desenvolver uma classe para agir como proxy da família de algoritmos de escalonamento. Outro estudo de caso ilustrado foi um servidor de vídeo adaptativo, com funcionalidades para atualizar dinamicamente a taxa de quadros e adicionar ou remover dinamicamente uma camada de compressão. Em ambos os casos, constatou-se Adapta minimiza o trabalho do desenvolvedor.

Considerando as experiências obtidas com o AutoGrid e o servidor de vídeo adaptativo, verificou-se que o Adapta facilitou a adição de comportamento adaptativo nesses dois projetos. Cabe ao desenvolvedor apenas a descrição dos aspectos de reconfiguração na linguagem **AdaptaML** e alteração de algumas das classes que compõem sistemas de software como, por exemplo, a modificação efetuada no GRM para recalcular a quantidade de réplicas a serem geradas para cada submissão ou a criação de dois proxies no servidor de vídeo adaptativo, um proxy para a família de objetos que envia quadros e outro proxy para a família que recebe os quadros.

Dentre os requisitos que puderam ser avaliados em face dos cenários analisa-

dos, encontra-se a definição de mecanismos de reconfiguração genéricos que possam ser utilizados em uma grande variedade de aplicações. Cita-se, como exemplo, a atualização dinâmica de parâmetros, usada tanto no AutoGrid quanto no servidor de vídeo adaptativo. Verificou-se, na introdução dinâmica do algoritmo de compressão no servidor de vídeo adaptativo, que o mecanismo de adaptação permitiu gerenciar as dependências entre os componentes de uma aplicação distribuída e assegurar a consistência e integridade de uma aplicação distribuída usando um protocolo de sincronização entre servidor e clientes.

Apesar dos esforços de avaliação descritos neste capítulo, alguns dos requisitos apresentados não puderam ser avaliados diante dos cenários propostos como, por exemplo, a flexibilidade da infra-estrutura de monitoramento para adicionar ou remover monitores e a capacidade de definição de eventos compostos oriundos da combinação de duas ou mais fontes. No entanto, trabalhos em desenvolvimento no Laboratório de Sistemas Distribuídos estão ampliando o uso framework em cenários mais complexos onde estes recursos serão utilizados.

5 Trabalhos Relacionados

Existem atualmente diversos projetos relacionados ao desenvolvimento de software auto-adaptativo, capaz de se reconfigurar dinamicamente diante de mudanças em seu ambiente de execução. Este capítulo descreve relevantes projetos no âmbito de software auto-adaptativo que possuem estreita relação com o objetivo do trabalho apresentado nesta dissertação.

5.1 Accord

O Accord [30,31,44] é um arcabouço para desenvolvimento de aplicações autônomas (auto-gerenciáveis) em ambientes distribuídos. O Accord faz parte do projeto AutoMate [45] cujo objetivo é o desenvolvimento de modelos conceituais e arquiteturas para concepção e execução de aplicações auto-gerenciáveis na grade computacional, minimizando a complexidade, heterogeneidade, dinamismo e incerteza inerentes a esse ambiente.

O Accord baseia-se na separação dos aspectos de composição dos elementos (organização, interação e coordenação) em relação ao comportamento computacional (funcional e não-funcional) do elemento. Essa separação de contextos permite duas formas de adaptação na arquitetura do Accord: uma ocorre em nível de elemento, através da execução de regras de alto-nível que modificam o comportamento e o estado do elemento em tempo de execução; enquanto a outra dá-se em nível de aplicação a partir da alteração de topologias, paradigmas de comunicação entre elementos, e modelos de coordenação da aplicação.

O arcabouço do Accord consiste em quatro conceitos: o contexto da aplicação, a definição de elementos autônomos, a definição de regras e mecanismos para composição dinâmica de elementos autônomos e uma infra-estrutura de agentes para assegurar autogerenciamento e composição dinâmica.

O contexto da aplicação compreende a descrição de espaços de nomes, sensores, atuadores, interfaces funcionais e eventos que permitem a interação e o entendimento entre elementos autônomos. A utilização de um contexto comum a todas as aplicações

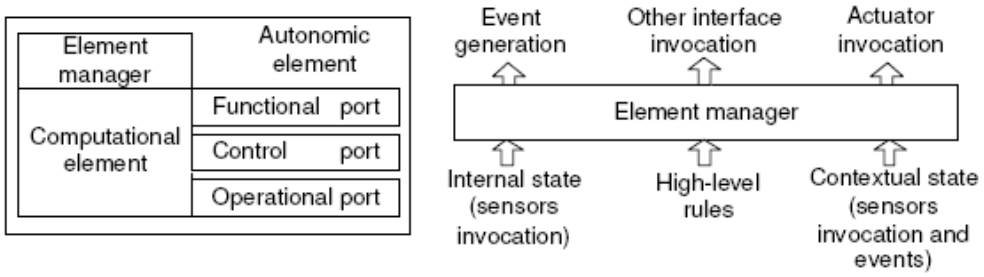

Figura 5.1: Elemento Autônomo e o Gerenciador de Elementos

permite que sejam definidas regras para gerenciamento autônomo do elemento e composição dinâmica e interação entre elementos.

O elemento autônomo é definido como o bloco-básico para desenvolvimento de aplicações auto-gerenciáveis. Ele estende objetos, componentes e serviços, definindo uma unidade de software com uma interface específica. Adicionalmente, um elemento autônomo encapsula regras, restrições e mecanismos para auto-gerenciamento e dinamicamente interage com outros elementos autônomos. A Figura 5.1 ilustra um elemento autônomo, que consiste em: uma **porta funcional**, que define o conjunto de funcionalidades providas e usadas pelo elemento; uma **porta de controle**, que define um conjunto de sensores e atuadores exportados pelo elemento e que, respectivamente, capturam e modificam o estado do mesmo; e uma **porta operacional** que define as interfaces para formulação, injeção e gerenciamento de regras.

As regras podem ser de dois tipos: de comportamento e de interação. As regras de comportamento controlam requisitos funcionais e não-funcionais da aplicação (ex: algoritmos, representação e formatação de dados). Enquanto isso, as regras de interação regem as interações entre elementos, entre elementos e o ambiente, e a coordenação entre os elementos que compõem uma aplicação autônoma. Por exemplo, uma regra de interação pode definir de onde o elemento obtém os dados de entrada e para onde encaminha as respostas das requisições, ou especificar os mecanismos de comunicação usados.

Cada elemento é associado a um Gerenciador de Elemento, a quem é delegada a execução do elemento. Ao Gerenciador de Elemento compete as seguintes funções: monitoramento do estado interno do elemento através de sensores; monitoramento do seu ambiente de execução; controle da execução das regras; produção de eventos; e realização invocações em outras interfaces ou sobre o próprio elemento através de atuadores.

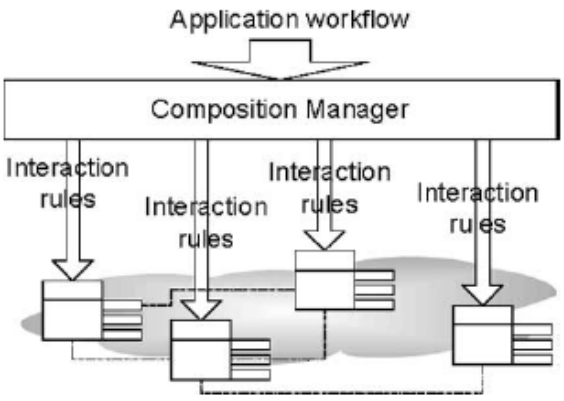

Figura 5.2: Gerenciamento de Interações

A composição dinâmica de elementos autônomos consiste na definição de uma organização de elementos e interações entre eles. A organização de elementos baseia-se na composição de portas funcionais. Em um arquivo XML, carregado no momento de carga da aplicação, o usuário do arcabouço descreve fluxos de trabalho que formam um grafo. Neste grafo os nós representam elementos autônomos e as arestas interações entre eles. O processo de composição, ilustrado na Figura 5.2, consiste na submissão do fluxo de trabalho, que é decomposto pelo Gerenciador de Composição e inserido nos respectivos elementos autônomos usando uma infra-estrutura de agentes. Em tempo de execução, o Gerenciador de Composição obtém as estratégias de adaptação e novos requisitos das aplicações e injeta regras (de comportamento e interação) nos elementos, através do Gerenciador de Elementos respectivo.

As adaptações no comportamento de um elemento são realizadas pelo Gerenciador de Elementos respectivo e incluem apenas aspectos não-funcionais como, por exemplo, tempo de execução, consumo de memória, largura de banda. Por outro lado, adaptações na composição dos elementos são iniciadas pelo Gerenciador de Composição e compreendem a adição, remoção e substituição de elementos definidos no fluxo de trabalho e das interações entre elementos. A transferência de estado e do conjunto de regras é realizada pelo Gerenciador de Composições que assegura a consistência do estado interno do elemento e a integridade das interações com os demais. O processo de adaptação ocorre em tempo de execução, sem re-compilação do código nem interrupção do serviço.

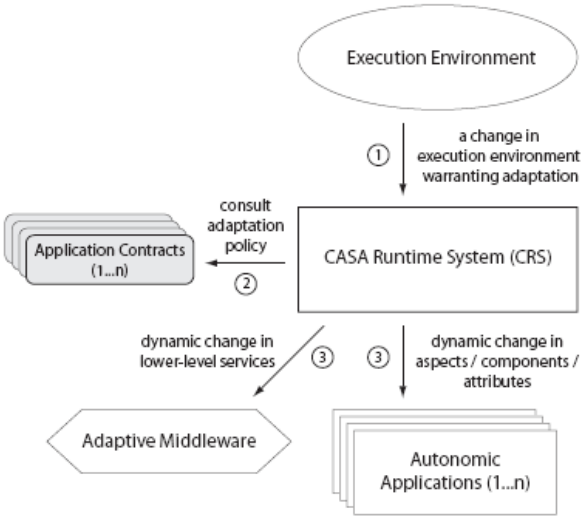

Figura 5.3: Processo de Adaptação no CASA

5.2 CASA

CASA (`Contract-based Adaptive Software Architecture`) [37–39] é um arcabouço para o desenvolvimento e operação de aplicações autônomas, desenvolvidas à luz do princípio de separação de responsabilidades: o código responsável pela adaptação é separado do código referente às regras de negócio. Além disso, CASA é um sistema de suporte a execução que realiza o monitoramento do ambiente computacional e inicia o processo de adaptação da aplicação assim que detectar mudanças significativas no ambiente computacional. As políticas de adaptação de uma aplicação são especificadas usando um contrato escrito em XML.

Cada nó computacional que hospeda uma aplicação autônoma executa o sistema de suporte a execução do CASA (CRS - *Casa Runtime System*). Basicamente, o processo de adaptação envolve três passos, ilustrados na Figura 5.3: o CRS detecta uma mudança no ambiente de execução que requer ações adaptativas; o CRS consulta os contratos das aplicações em relação à mudança de estado; e o CRS conduz a adaptação segundo a política especificada no contrato da aplicação.

O CRS consiste em quatro componentes, delimitados pela área pontilhada na Figura 5.4: RM (Gerenciador de Recursos), o CM (Monitor de Contexto), o AAS (Sistema de Adaptação de Aspectos) e o CAS (Sistema de Adaptação de Componentes). Estes quatro componentes promovem o monitoramento do ambiente computacional e a adoção de políticas de adaptação nas aplicações autônomas segundo o contrato expresso

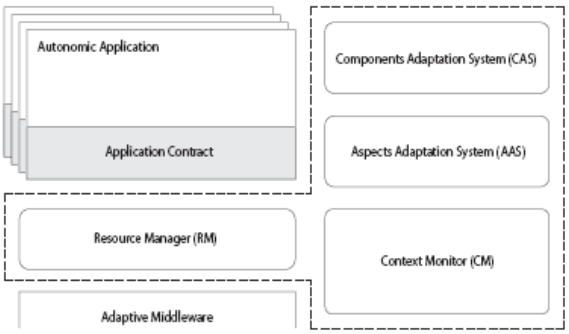

Figura 5.4: Arquitetura do CASA

pelo usuário do arcabouço.

O monitoramento do ambiente computacional compreende informações contextuais e recursos computacionais. A coleta de informações contextuais é realizada pelo CM através de sensores que obtêm os dados de contexto e os submetem a interpretadores, cuja função é estruturá-los em ontologias. Estas são submetidas a um analisador que irá extrair o conhecimento final relevante à aplicação. Exemplos de informações contextuais são: a localização do usuário e a identidade de objetos próximos com os quais se pode interagir). Enquanto isso, o monitoramento da disponibilidade de recursos computacionais é realizado por serviços de monitoramento externos (por exemplo, DProc [2] ou Remos [32]) que interagem com o RM exportando os dados coletados.

As alterações relevantes nas informações de contexto e variações na disponibilidade de recursos iniciam o processo de adaptação da aplicação. O CASA compreende quatro políticas de adaptação:

- Mudança dinâmica nos serviços de baixo-nível, que são os serviços essenciais para execução de aplicações. Por exemplo: transmissão de dados, compressão, codificação e decodificação de mídias. Esta adaptação é realizada por plataformas de middleware reflexivo que podem ser integradas ao CASA utilizando funcionalidades do RM, a quem compete iniciar o processo de adaptação;

- Introdução e remoção dinâmica de aspectos, também chamado POA dinâmica, que permite a adaptação de funcionalidades entrelaçadas no código da aplicação sem que se afete o código responsável pelas regras de negócio. Por exemplo, o comportamento de persistência pode ser modificado em razão da perda de conectividade com um armazém estável. O CASA utiliza o arcabouço PROSE [41] em coordenação com

o AAS nesse mecanismo de adaptação. O AAS submete o nome e a localização do arquivo do aspecto ao PROSE. Neste arquivo estão contidos os pontos de junção e o código binário do aspecto. O PROSE é capaz de interceptar pontos de junção em uma aplicação Java e invocar o código binário nesses pontos;

- Recomposição dinâmica de componentes da aplicação, que permite a adaptação do código responsável pelas regras de negócio. Uma ação de recomposição dinâmica é gerenciada pelo CAS e consiste na adição, remoção ou substituição de componentes em tempo de execução. Ao CAS compete assegurar a consistência de toda a aplicação autônoma, através da transferência de estado entre os componentes e manutenção da integridade das conexões estabelecidas antes do início da adaptação; e

- Mudança dinâmica nos atributos da aplicação, tais como o valor de *timeout* ou a frequência de transmissão de dados. Para que essa adaptação ocorra, os desenvolvedores precisam introduzir nos códigos das aplicações *callbacks*, que são invocados em tempo de execução pelo RM.

A política de adaptação de cada aplicação é definida em um contrato, escrito em XML que é externo à aplicação. Isso facilita sua modificação, extensão e padronização em tempo de execução. O contrato é divido em elementos `<context>`, onde cada um deles representa o estado de informação contextual de interesse da aplicação. Os parâmetros que caracterizam este estado são especificados pelo elemento `<params>`. Cada parâmetro contém elementos `<par>`, constituídos por um par de atributos nome e valor (**name** e **value**, respectivamente).

Cada `<context>` contém a lista de configurações alternativas da aplicação, que variam segundo os requisitos de recursos computacionais. Cada elemento `<config>` representa uma configuração, com requisitos de recurso (`<resource>`), componentes adaptáveis e aspectos constituintes (`<componentes>` e `<aspects>`), métodos *callback* (`<callbacks>`) e serviços de baixo-nível (`<llservices>`). Todos os elementos de uma configuração são opcionais e descrevem as ações que devem ser tomadas em cada configuração.

CASA provê um protocolo de negociação necessário quando, em uma aplicação colaborativa, uma determinada configuração é selecionada e precisa ser aprovada pelas aplicações que colaboram. O Coordenador de Serviços (SC) coordena este protocolo de

negociação. O SC coleta, junto ao RM, todas as configurações válidas e as informa a todas as aplicações que irão ordenar as configurações segundo suas preferências. Com base no ranking realizado, o SC escolhe a configuração mais apropriada. Considere por exemplo uma aplicação que transmite mídias de vídeo e aúdio em alta-qualidade. Esta aplicação pode se reconfigurar dinamicamente em resposta a variações na largura de banda, através da redução da qualidade do aúdio e manutenção da alta-qualidade do vídeo ou vice-versa. Antes de efetuar a reconfiguração, é realizado o protocolo de negociação com os clientes da aplicação. A negociação irá depender do conteúdo do arquivo de mídia: se este for um jogo de futebol os clientes irão optar por preservar a alta-qualidade do vídeo; caso seja um concerto musical, os clientes selecionarão manter a alta-qualidade do áudio.

5.3 QuO

QuO [63] é um arcabouço para criação de aplicações distribuídas que podem se adaptar a mudanças na qualidade de serviço (QoS). Os objetivos de QuO são: criar aplicações distribuídas mais robustas, que executem eficientemente sobre uma ampla variedade de condições do ambiente operacional; dar maior controle sobre o comportamento adaptativo das aplicações em todo o seu ciclo de vida; suporte a reutilização de mecanismos adaptativos entre aplicações com diferentes interfaces e funcionalidades.

Em aplicações CORBA tradicionais, o cliente faz a invocação de um método em um objeto remoto através de sua interface. A chamada é processada pelo ORB na máquina cliente, entregue ao ORB na máquina servidora e processada pelo objeto remoto. Uma aplicação QuO adiciona etapas adicionais a este processo. O sistema de suporte a execução do QuO mede o estado de QoS do sistema a cada invocação ou retorno de chamada, detecta quando as condições de sistema mudaram, inicia alterações de comportamento e dispara *callbacks*. O desenvolvedor da aplicação adaptativa usa uma linguagem de descrição de QoS chamada QDL para definir como a aplicação vai se adaptar. O QDL permite descrever estados de QoS, comportamentos adaptativos e condições de sistema que precisam ser medidas e controladas.

Geradores de código interpretam o conteúdo descrito em QDL para produzir código que é ligado com o código do cliente, o kernel QuO e com os *stubs* e *skeletons* de CORBA. O código resultante implementa os seguintes componentes: contratos, que

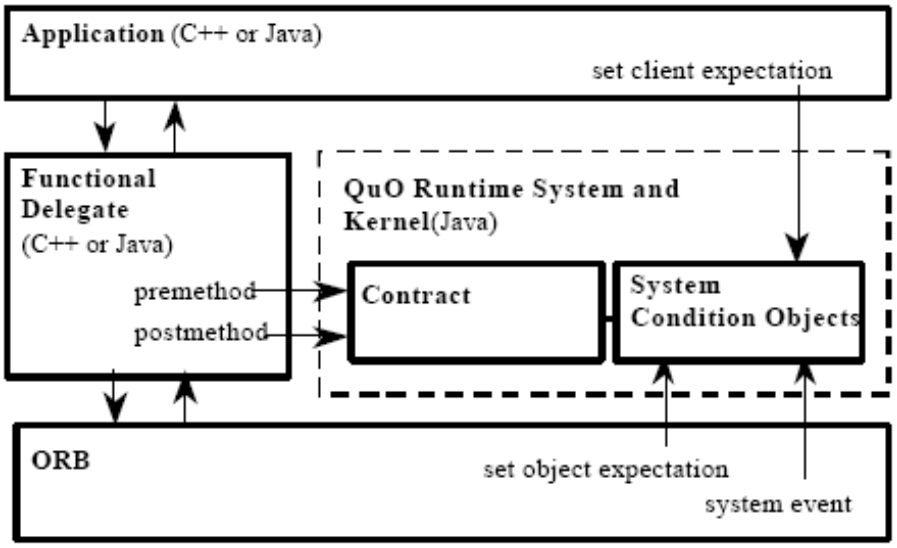

Figura 5.5: Arquitetura QuO

armazenam o estado de QoS do sistema e disparam comportamentos adaptativos quando ocorrem mudanças de estado; delegados, que possuem interfaces idênticas aos clientes e objetos remotos, contendo código adicional para verificar o contrato e selecionar qual adaptação é mais adequada; e condições de sistema, que são usadas para medir e controlar QoS.

Em QuO, o mapeamento CORBA de um cliente a um objeto remoto é substituído por uma conexão de um delegado até o objeto. Quando um cliente realiza uma chamada ao objeto remoto, ele está na verdade fazendo uma chamada ao delegado local. Opcionalmente, o delegado obtém os valores das condições de sistema e realiza a avaliação do contrato. A arquitetura de QuO divide o ambiente em um delegado, que executa junto à aplicação, e contratos e condições de sistema, que executam em uma instância do kernel QuO.

A Figura 5.5 ilustra os componentes do QuO e suas interações. O **kernel QuO** é a biblioteca de serviços essenciais para o sistema de suporte a execução, os contratos e as condições de sistema. O kernel provê dois serviços essenciais:

- Uma **fábrica de objetos**, que cria e inicializa contratos e condições de sistemas em tempo de carga, usando reflexão computacional. Neste processo, a fábrica recebe uma invocação com o nome da classe para criação como parâmetro, obtém o código binário da classe e efetua a carga dinâmica (se já não estiver carregada); em seguida, um novo objeto é instanciado e a referência retornada a quem enviou a solicitação;

- Um **avaliador de contratos**, que pode ser invocado no pré-método ou pós-método de um delegado ou quando ocorrer a alteração de uma condição de sistema monitorada.

O kernel QuO é flexível em relação ao seu ambiente de execução. Ele pode executar em um processo separado ou no mesmo processo da aplicação; ele pode executar no mesmo nó computacional da aplicação ou em nó diverso. Além disso, diversas aplicações podem compartilhar um mesmo kernel, se necessário.

Os **delegados** encapsulam objetos remotos (usando o Padrão Wrapper [6]) com um comportamento que permite a adaptação em razão de mudanças na QoS. Ainda que a interface do delegado seja idêntica à do objeto CORBA encapsulado, o código da aplicação cliente ainda exige algumas modificações para estabelecer a conexão com o delegado. No código cliente, deve existir uma chamada ao método `connect`, que é usado para estabelecer conexões aos objetos QuO necessários. Essa chamada busca por uma instância do kernel QuO ou inicializa uma nova. Em seguida, invoca a fábrica de objetos do kernel para criar os objetos de contrato e condições de sistema. Finalmente, são estabelecidas as ligações entre os objetos.

Os **contratos** coletam dados das condições de sistema e os analisam para determinar o estado de QoS atual. Cada contrato consiste em uma ou mais regiões, definidas por um predicado sobre alguns valores nas condições de sistema. Quando o predicado for verdadeiro, a região encontra-se ativa. Mais de uma região pode estar ativa em um dado momento. Porém, como a avaliação é sequencial, apenas o primeiro predicado que for avaliado como verdadeiro é a região atual. Uma transição de regiões é definida quando a nova região atual é distinta da anterior. Nestes casos, pode ocorrer a chamada de métodos em condições de sistema para modificar a QoS ou invocações assíncronas a métodos *callback* para notificar mudanças de QoS.

As **condições de sistema** representam, individualmente, uma propriedade do sistema (por exemplo, número de processos ativos na máquina servidora ou o tempo de resposta de uma invocação). Cada condição de sistema implementa uma interface com o método `getValue`, usado pelos contratos para obter os valores das propriedades do sistema. De posse desses valores, o contrato pode determinar a região de QoS em que o sistema se encontra.

5.4 Adaptive.NET

O Adaptive.NET [47–49] é um arcabouço para reconfiguração dinâmica de aplicações adaptativas baseadas em componentes. Originalmente, o arcabouço foi desenvolvido baseado na plataforma .NET da Microsoft e é capaz de reconfigurar aplicações construídas com componentes proprietários .NET. Contudo, alguns dispositivos com recursos computacionais limitados não oferecem suporte a plataforma .NET, por exemplo, aqueles que não executam o sistema operacional Windows CE. Para esses dispositivos, o Adaptive.NET foi estendido para integrar componentes Java/CORBA, permitindo a adaptação dinâmica de aplicações heterogêneas. Por exemplo, pode-se ter uma aplicação cliente-servidor de exibição de vídeos, onde o servidor é desenvolvido usando componentes .NET e o cliente é desenvolvido com J2ME/CORBA.

A visão-geral da arquitetura está ilustrada na Figura 5.6. A infra-estrutura de reconfiguração, denominada CoFRA, é o componente principal do Adaptive.NET. O mecanismo de adaptação do CoFRA avalia políticas de reconfiguração, que são originadas do mapeamento entre parâmetros de monitoramento e configurações da aplicação. Por exemplo, modificar as conexões de um componente para usar um algoritmo de localização menos preciso, mas que consuma menos bateria do dispositivo móvel.

A infra-estrutura de monitoramento coleta três tipos de propriedades: condições do ambiente de execução, por exemplo: memória disponível, consumo de UCP, largura de banda; estado dos componentes da aplicação como, por exemplo, ciclo de vida de um componente ou falha de um componente; e propriedades de componentes como, por exemplo, algoritmos usados, contadores. O mecanismo de adaptação do CoFRA é responsável por interagir com os Configuradores de Componentes (CoCo) para obter os valores relacionados ao componente e com observadores específicos para obter informações do ambiente de execução (por exemplo: observador de bateria, observador de largura de banda).

Durante a inicialização de uma aplicação, o CoFRA avalia o perfil da mesma, inicia o monitoramento do ambiente computacional e carrega a configuração da aplicação mais adequada às condições do ambiente de execução. As configurações são descritas em um arquivo de descrição escrito em uma linguagem baseada em XML. Este documento compreende as informações dos componentes envolvidos, tais como as portas de comunicação, o mapeamento dos nós e o código binário de onde o componente será instanci-

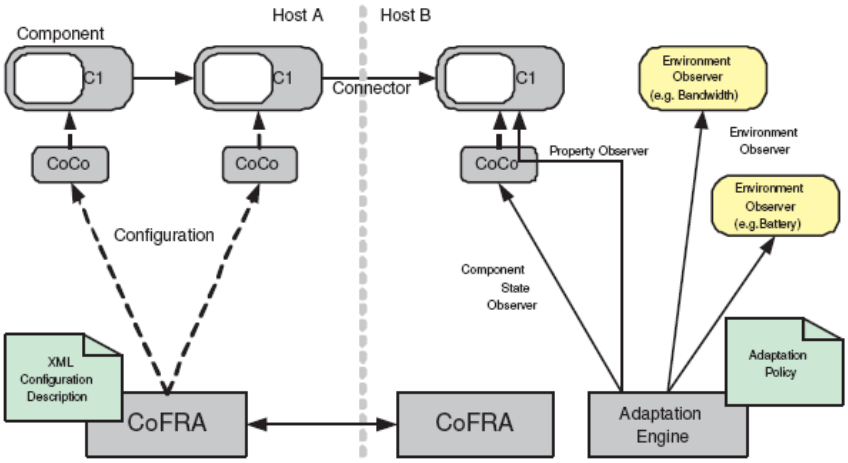

Figura 5.6: Arquitetura de Adaptação na Plataforma .NET

ado. Os componentes implementam uma interface específica de reconfiguração (chamada `IConfigure`), que contém métodos para conectar componentes, alterar propriedades dos componentes, bloquear conexões, iniciar e finalizar o processamento de um componente.

Os componentes podem ser instanciados e gerenciados em diversas plataformas, tais como Java, CORBA e .NET. Além disso, a instanciação pode se dar em processos independentes, em threads ou como simples objetos. Configuradores de componentes (CoCo) abstraem a complexidade no acesso a diversos tipos de componentes. Um CoCo possui métodos para criar componentes, acessar a interface de configuração, consultar o estado do componente e o seu ciclo de vida e remover a instância de um componente da aplicação. A fim de minimizar o arcabouço Adaptive.NET, os CoCo são instanciados dinamicamente pelo CoFRA na medida em que forem necessários. Desta forma, o espaço em memória consumido pelo Adaptive.NET é diminuído, permitindo a execução em dispositivos com menor poder computacional.

As ações de reconfiguração dinâmica previstas no CoFRA são a adição e remoção de componentes e a modificação dos atributos do componente. Outras operações podem ser especificadas a partir de um conjunto destas operações. Por exemplo, a substituição de um componente consiste na remoção do componente antigo e adição de um novo componente. Nenhuma das operações define um protocolo de transferência de estado. Antes do início da reconfiguração, o componente deve atingir um estado de inatividade, onde sejam bloqueadas as requisições enviadas e as demais requisições em processamento tenham sido concluídas. Ao término desse processo, as requisições bloqueadas são redire-

cionadas ao novo componente.

A comunicação entre componentes é realizada por conectores. O estabelecimento dessas conexões é realizado usando um método da interface IConfigure, que usa a API Reflection do .NET para este fim. No estado atual de implementação do Adaptive.NET, existem três tipos de conectores implementados: conectores de chamadas locais, que representam uma simples chamada a um método; conectores .NET Remoting; e conectores IIOP, usadas para comunicação com objetos CORBA. A interoperabilidade entre CORBA e .NET é realizada com o arcabouço de interoperação Janeva [5], que adiciona a camada do protocolo IIOP em cima de .NET e adiciona serviços CORBA e um ORB no arcabouço .NET. Posteriormente, a arquitetura pode ser estendida, incorporando, por exemplo, WebServices com conectores SOAP.

5.5 OpenRec

OpenRec [22, 23, 65] foi desenvolvido para permitir reconfiguração automática de aplicações baseadas em componentes. Uma das características do OpenRec é ser aberto e extensível com relação ao modelo de componentes e ao gerenciamento da reconfiguração. Dessa maneira, o OpenRec permite a modificação da arquitetura de uma aplicação, adicionando, removendo ou substituindo componentes e conexões; e, ao mesmo tempo, possibilita a modificação do próprio arcabouço, disponibilizando novos algoritmos de reconfiguração em tempo de execução. A arquitetura em três camadas está ilustrada na Figura 5.7 e compreende um Driver de Mudança, o Gerenciador de Reconfiguração e a camada de Aplicação.

O Driver de Mudança é um contâiner para os desenvolvedores de aplicações descreverem quando e quais mudanças são necessárias. A saída do Driver de Mudança é um arquivo no formato OpenRecML, uma linguagem baseada em XML, usada para descrever configurações de componentes e alterações em configurações existentes (reconfigurações).

O Gerenciador de Reconfiguração é também um contâiner que armazena uma implementação de um algoritmo de reconfiguração, que determina como a adaptação será realizada. Por exemplo, pode-se selecionar em tempo de execução qual algoritmo será utilizado para reconfigurar uma aplicação específica. A última camada, de Aplicação, é

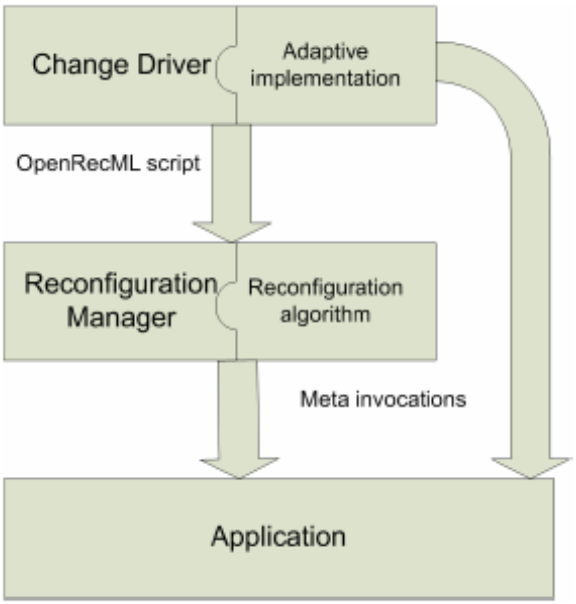

Figura 5.7: Arquitetura do OpenRec

um conjunto de componentes de aplicação escritos por desenvolvedores.

Cada camada do OpenRec é construída usando um modelo de componentes reflexivos. Os componentes são definidos como uma unidade de composição com interfaces bem-definidas e descrição explícita das interfaces necessárias, ilustrando quais os serviços que o componente espera de outros para satisfazer seu próprio contrato. Além disso, os componentes são transparentes em nível de interconexão, ou seja, eles não tem conhecimento de suas conexões com outros componentes. Por exemplo, para responder uma mensagem recebida o componente usa a interface definida sem conhecimento de qual componente iniciou o pedido. Ao mesmo tempo, para enviar uma requisição, o componente usa a interface necessária sem saber qual componente está conectado naquele momento. Esse desacoplamento de cada componente possibilita a adição, remoção, substituição e migração de componentes e alterações na topologia de conexões.

O modelo de componentes possui dois níveis: um meta-nível e um nível base. No OpenRec, o nível base corresponde aos componentes e interfaces da aplicação. No meta-nível encontram-se serviços para descobrir e modificar as configurações (reflexão estrutural) e inserir ou remover interceptadores e conectores (reflexão comportamental). Os interceptadores permitem que o Driver de Mudança possa monitorar o comportamento do componente e introduzir código a ser executado na comunicação entre componentes.

Os algoritmos de reconfiguração desenvolvidos para o OpenRec devem imple-

mentar a interface `IReconfigurationAlgorithm`. Esta interface disponibiliza o método `start()`, que consiste em sub-passos:

1. `doCheckConstraints()`, que verifica as restrições do ambiente computacional e se é possível prosseguir com a reconfiguração;
2. `doSynchronise()`, que implementa as ações para sincronizar a aplicação, por exemplo, o bloqueio de interfaces prevenindo o início de novas comunicações; e
3. `doReconfigure()`, que efetua as mudanças estruturais necessárias na reconfiguração, invocando operações de adaptação definidas nos meta-objetos da aplicação.

OpenRec permite a coexistência de várias técnicas de sincronização durante ações de reconfiguração, usando o padrão Strategy [6]. `IReconfigurationAlgorithm` é a interface da estratégia e os algoritmos de reconfiguração são as estratégias que podem ser selecionadas dinamicamente. O usuário do arcabouço ou o administrador da aplicação pode observar o comportamento de diferentes algoritmos e estimar o custo de execução de cada um, medidos em termos de: quantidade de componentes afetados, tempo total para término da reconfiguração, dentre outros. Usando estas informações, pode-se selecionar em tempo de execução o algoritmo mais adequado para determinada aplicação ou condição do ambiente operacional.

Finalmente, o OpenRec busca assegurar a manutenção da integridade de uma aplicação. Por exemplo, em um servidor de áudio, a adição de um componente de codificação do lado servidor sem a adição de um componente de decodificação do lado cliente causa mau-funcionamento na execução de mídias. Esta ação de sincronização está contida na lógica de cada um dos algoritmos de reconfiguração.

5.6 Draco

Draco (Distrinet Reliable and Adaptive Components) [62] é um ambiente em execução modular e extensível, desenvolvido para computação pervasiva parcialmente aderente à filosofia de componentes Seescoa [51], baseada na modificação do paradigma de desenvolvimento baseado em componentes para se adaptar as necessidades de softwares embutidos. Os conceitos da metodologia Seescoa implementados pelo Draco são:

- **Planta do componente**, uma entidade reutilizável que contém a descrição e implementação de um componente. Não existe em tempo de execução;
- **Componente**, usado como elemento para construção de aplicações. Consiste na instanciação de uma planta de componente e tem existência em tempo de execução e estado;
- **Porta**, mecanismo de comunicação entre componentes. Todo componente possui zero, uma ou mais portas associadas;
- **Conector**, conexão funcional entre componentes, permitindo o envio e recebimento de mensagens; e
- **Contrato**, que impõe restrições não-funcionais (por exemplo: uso de memória ou tempo de resposta) em um componente ou grupo de componentes.

A arquitetura do Draco é apresentada na Figura 5.8. O núcleo do sistema compreende 5 unidades. Em tempo de carga, o núcleo do sistema é montado dinamicamente usando o padrão Builder [6], que lê de um arquivo de descrição qual implementação usar para as unidades do Draco. Uma vez instanciado, o núcleo do sistema é considerado fixo.

O Gerenciador de Componentes é uma fábrica de componentes e portas, que mantém um repositório das referências de componentes instanciados no sistema. O Escalonador é responsável por escalonar mensagens para envio, assegurando a correta ordenação das mensagens. O Gerenciador de Mensagens é responsável pela entrega das mensagens. Para isso, ele precisa recuperar o conector associado à porta que originou a mensagem. O Gerenciador de Conectores cria e mantém conectores entre portas. Gerenciador de Módulos estende o sistema em execução do Draco com módulos adicionais, que podem ser carregados e descarregados em tempo de execução.

Uma das extensões do Draco é a atualização dinâmica de aplicações. Para substituir de modo seguro um componente em tempo de execução, Draco segue alguns passos:

1. O componente é posto em um estado de inatividade, com o bloqueio de todas as novas requisições para o componente;
2. Uma nova planta do componente é instanciada usando o Gerenciador de Componentes. As portas também são criadas;

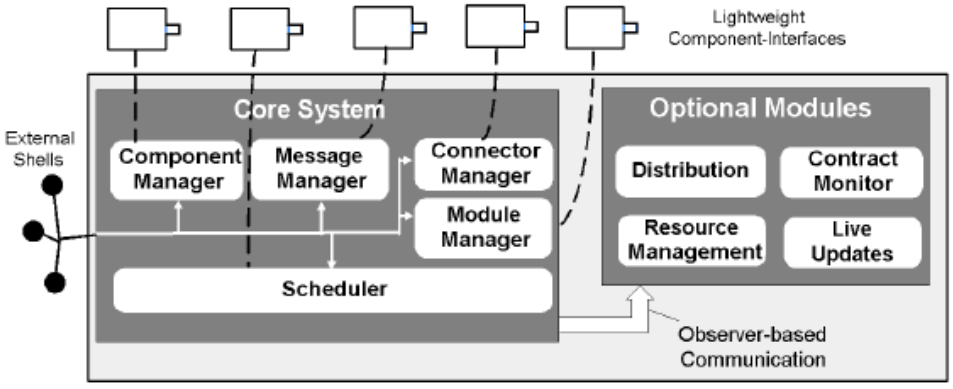

Figura 5.8: Componentes do Núcleo do Draco

3. O estado da versão antiga do componente é transferido para a nova versão;
4. Os conectores são re-conectados após requisição ao Gerenciador de Conexões;
5. O novo componente é ativado; e
6. O componente antigo é removido.

5.7 Comparação

Esta seção apresenta uma análise comparativa dos principais trabalhos apresentados em relação ao Adapta. Tomamos como base a taxonomia de McKinley, Sadjadi, Kasten e Cheng 2.2.3, respondendo as perguntas propostas: *como?*, *quando?* e *onde?*. Desta maneira, obtivemos sete critérios de comparação:

1. A **técnica** utilizado, que pode ser: reflexão computacional, programação orientada a aspectos e padrões computacionais;
2. A **transparência** com relação ao código funcional da aplicação, observando se esse precisa ser modificado para adicionar comportamento adaptativo;
3. A **granularidade**, que pode ser de objeto, componente e serviço, dentre outras;
4. As ferramentas de **suporte** utilizadas;
5. O **tempo** em que se dá a adaptação: tempo de carga ou de execução;
6. O **lugar** em que se dá a adaptação: na aplicação ou na camada de serviços comuns do middleware.

A Tabela 5.1 apresenta o resultado da comparação efetuada. O Adapta usa mecanismos de reflexão computacional para realizar as ações de adaptação nas aplicações e o padrão Adaptive-Object Model para estruturar o arcabouço de forma a permitir que o mesmo seja alterado dinamicamente sem interrupção do serviço. O Adaptive.NET, CASA e OpenRec também usam mecanismos de reflexão computacional para obter auto-conhecimento e adaptabilidade da aplicação. O CASA ainda usa técnicas de programação orientada a aspectos, a partir da integração com o arcabouço PROSE. Alguns outros trabalhos usam padrões de projeto: Draco usa o padrão Builder para, em tempo de carga, montar o núcleo do arcabouço dinamicamente; OpenRec usa o padrão Strategy para substituir o algoritmo de reconfiguração usado no arcabouço em tempo de execução; e o QuO usa o padrão Wrapper para criar uma camada de indireção sobre o objeto remoto. QuO ainda usa técnicas de integração de middleware, que envolve realização da adaptação na camada de serviços comuns. O CASA faz uso dessas mesmas técnicas, mas a adaptação é realizada por um middleware reflexivo e não pelo arcabouço, que apenas coordena o processo.

Observa-se ainda na tabela que nenhum dos arcabouços é transparente em relação ao código funcional da adaptação. Desta forma, o desenvolvedor de aplicações deverá sempre escrever código adicional para que a aplicação se torne adaptativa. Por exemplo, no Adapta o desenvolvedor deve estender de uma classe específica para criar um parâmetro atualizável ou escrever o código do proxy de uma família de objetos. Apenas o CASA pode, eventualmente, dispor de transparência, desde que ele baseie a reconfiguração de uma aplicação inteiramente em serviços de baixo nível. Para isso, o middleware reflexivo que for integrado deve efetuar a adaptação transparentemente.

Nas soluções descritas, a granularidade da mudança dá-se quase sempre em nível de componente (Adaptive.NET, CASA, Draco, OpenRec). Isso porque a principal operação de reconfiguração nesses arcabouços envolve a recomposição dinâmica de aplicações. Por exemplo: adicionar, remover ou substituir componentes e conexões. No Adapta, no QuO e também no CASA a granularidade dá-se em nível de objeto. No Accord, a granularidade dá-se em nível de elemento, entidade concebida nesse arcabouço que consiste em portas funcionais, operacionais e de controle. Um elemento pode ser um objeto, componente ou mesmo serviço.

No suporte, observa-se que a maioria dos arcabouços usa o middleware CORBA como infra-estrutura de suporte para execução de aplicações adaptativas, por exemplo:

Adapta e QuO. O Adaptive.NET destaca-se porque permite a interoperabilidade do arcabouço em CORBA e .NET, possibilitando a reconfiguração de aplicações heterogêneas cujos componentes encontram-se em plataformas de middleware distintas. O Accord está escrito para execução na plataforma AutoMate, que oferece serviços comuns de middleware como serviço de nomes ou serviço de ciclo de vida.

Em relação ao tempo de adaptação, quase todas as soluções permitem reconfiguração dinâmica, ou seja, em tempo de execução. Isso significa que o sistema não é interrompido durante a reconfiguração. No Adapta, todas as ações de reconfiguração das aplicações e do próprio arcabouço dão-se em tempo de execução. Apenas o QuO não tem soluções em tempo de execução.

A última dimensão de análise é o local em que o código adaptativo é inserido. Quase todas as abordagens inserem o código na camada da aplicação ou através de um meta-nível que dinamicamente modifica os objetos no nível base (por exemplo: Adapta, CASA) ou estendendo interfaces especializadas em reconfiguração (por exemplo: `IConfigure` do Adaptive.NET ou `IReconfigurationAlgorithm` do OpenRec). O CASA e o QuO fazem também atualização em nível da camada de serviços comuns do middleware, modificando serviços de baixo nível usados por todas as aplicações.

Projeto	Técnica	Transparência para o Desenvolvedor	Granularidade	Suporte	Tempo	Local
Adapta	Reflexão computacional / Padrão Adaptive-Object Model	Não	Objeto	CORBA	Tempo de execução	Aplicação
Accord		Não	Elemento[a]	AutoMate	Tempo de execução	Aplicação
Adaptive.NET	Reflexão computacional	Não	Componente	.NET / CORBA	Tempo de execução	Aplicação
CASA	Reflexão computacional / Programação Orientada a Aspectos / Integração de middleware	Não[b]	Objeto / Componente / Serviço		Tempo de execução	Aplicação e Camada de serviços comuns (middleware)
Draco	Padrão Builder	Não	Componente		Tempo de execução	Aplicação
OpenRec	Reflexão computacional / Padrão Strategy	Não	Componente		Tempo de execução	Aplicação
QuO	Integração de middleware / Padrão Wrapper	Não	Objeto	CORBA	Tempo de compilação, ligação e carga	Camada de serviços comuns (middleware)

Tabela 5.1: Resumo comparativo dos principais trabalhos relacionados

[a] Definição que compreende objetos, componentes e serviços
[b] A adaptação dos serviços de baixo-nível é transparente

6 Conclusão e Trabalhos Futuros

Ambientes computacionais modernos caracterizam-se pela heterogeneidade de dispositivos computacionais e da infra-estrutura de computação. Em face deste cenário, o desenvolvimento de novas aplicações tem se tornado árduo, pois as aplicações estão voltadas para um conjunto mais amplo de dispositivos computacionais, que variam desde servidores de alta capacidade a dispositivos portáteis com recurso limitados. Outra característica desses ambientes é o acentuado grau de dinamismo dos sistemas e aplicações, responsável por modificações abruptas e imprevisíveis no ambiente de execução. Tudo isso motiva o desenvolvimento de aplicações capazes de alterar sua estrutura e funcionalidade automaticamente, sem re-compilação do código ou intervenção do desenvolvedor e transparentemente, sem interrupção do serviço provido. Estas aplicações são usualmente denominadas aplicações adaptativas.

Este trabalho apresentou o Adapta, um arcabouço para desenvolvimento de aplicações distribuídas adaptativas baseado em reflexão computacional, dividindo o software em dois níveis: no meta-nível encontram-se informações sobre propriedades do software, tornando-o autoconsciente de seus elementos e estrutura; e, no nível-base, a lógica da aplicação, as regras de negócio. As mudanças efetuadas no meta-nível afetam o comportamento dos objetos no nível base. O Adapta também é um sistema de suporte à execução de aplicações adaptativas, responsável pelo monitoramento do ambiente de execução, detecção e notificação de mudanças significativas na disponibilidade de recursos. O Adapta ainda introduz a **AdaptaML**, uma linguagem bem estruturada de reconfiguração, que permite a definição dos recursos a serem monitorados, os eventos a serem gerados em face de mudanças no ambiente de execução, os elementos adaptáveis da aplicação e as ações de reconfiguração aplicáveis ao nível base.

As principais contribuições deste trabalho foram:

- A definição de uma arquitetura na qual o monitoramento do ambiente computacional, a detecção e notificação de eventos e o mecanismo responsável pela reconfiguração dinâmica da aplicação estão desacoplados em componentes distintos:
  - Serviço de Monitoramento, responsável por coletar periodicamente os dados

do ambiente de execução em um determinado nó computacional e notificar mudanças significativas na disponibilidade dos recursos monitorados;

  - Serviço de Eventos Locais, responsável por detectar eventos originados em um único nó e notificá-los a aplicações e componentes registrados, no momento em que uma condição, expressa através de uma sentença lógica for atingida;
  - Serviço de Eventos Distribuído, responsável por combinar as notificações de eventos locais ao longo da rede, possibilitando a tomada de decisões de reconfiguração envolvendo mais de um nó computacional; e
  - Serviço de Reconfiguração Dinâmica (DyReS), compreendendo o mecanismo de adaptação da arquitetura, responsável por disparar ações de reconfiguração nos componentes da aplicação em resposta a mudanças no ambiente de execução.

- O desenvolvimento de um sistema de suporte a execução de aplicações adaptativas, gerenciado pelo usuário do arcabouço e presente em cada nó computacional que possa conter aplicações adaptativas, que consiste em algumas operações básicas: interrupção ou prosseguimento de monitoramento de um recurso específico; adição ou remoção dinâmica de monitores no sistema; suspensão ou ativação da detecção de um evento.

- A concepção de uma linguagem bem-estrutura de reconfiguração, chamada **AdaptaML**, escrita a partir de XML e que permite a definição de: recursos computacionais monitoráveis; eventos locais e compostos, que serão detectados e notificados; elementos adaptáveis da aplicação (parâmetros, famílias de objetos e componentes); e ações de reconfiguração a serem aplicadas no nível-base da aplicação adaptativa.

- A implementação de um interpretador do arquivo de reconfiguração **AdaptaML**, com a carga do conteúdo do arquivo dinamicamente na aplicação, sem modificação de código ou interrupção do serviço.

- A definição de uma arquitetura de middleware de grade autônomo chamada AutoGrid, com mecanismos de ciência de contexto, auto-configuração, auto-cura e auto-otimização.

- O desenvolvimento de um Servidor de Vídeo Adaptativo, com operações de modificação dinâmica da taxa de quadros transmitidos e adição dinâmica de uma camada de compressão de dados antes do envio dos quadros aos clientes.

Ressalta-se que, durante o desenvolvimento deste trabalho, gerou-se três publicações:

- *Adapta: A framework for dynamic reconfiguration of distributed applications* [53]: pôster publicado no *Proceedings of the 5th workshop on Adaptive and reflective middleware (ARM '06)*, que descreve a arquitetura do Adapta;
- *The Adapta Framework for Building Self-Adaptive Distributed Applications* [54]: artigo completo publicado no *The Third International Conference on Autonomic and Autonomous Systems (ICAS 2007)*, que descreve a arquitetura do Adapta, seus principais componentes e aspectos de sua implementação;
- *AutoGrid: Towards an Autonomic Grid Middleware* [55]: artigo completo publicado no *Fourth International Workshop on Emerging Technologies for Next-generation GRID (ETNGRID 2007)*, que descreve a arquitetura de uma grade autônoma (AutoGrid) que usa a infra-estrutura do Adapta.

6.1 Trabalhos Futuros

Com o avanço das pesquisas para o desenvolvimento deste trabalho, identificamos diversas oportunidades de extensão, entre as quais, destacamos:

- Modificar o Serviço de Eventos Locais e o Serviço de Eventos Distribuído substituindo o uso de canais de eventos CORBA, que fazem uso de técnicas não-confiáveis, por outro mecanismo para produção e consumo de eventos;
- Estender as ações de reconfiguração providas pelo arcabouço com o suporte a reconfiguração de componentes inteiros, a partir de ações de adição, remoção, substituição com transferência de estado, migração entre nós na rede computacional e replicação;
- Desenvolver uma sintaxe usando XML Schema para validação da linguagem de reconfiguração do arcabouço;
- Implementar uma interface gráfica para que o usuário do arcabouço descreva o modelo de dados dos componentes do arcabouço, com a introdução, remoção e alteração de elementos do modelo de dados sendo realizadas visualmente;

- Implementar um mecanismo de seleção da técnica de recuperação mais adequada ao ambiente operacional da grade AutoGrid, com base nos parâmetros de tolerância a falhas, tais como: tempo médio entre falhas (MTBF), tempo de execução sem falhas, dentre outros.

- Implementar o mecanismo de suporte a substituição do algoritmo de escalonamento usado no AutoGrid, incluindo a análise do impacto das variáveis de estado do ambiente computacional (taxa de chegada de aplicações, taxa de ocupação de recursos, prioridade de execução de uma aplicação) na seleção do algoritmo mais satisfatório em um determinado momento de execução; e

- Implementar o mecanismo de modificação dinâmica da resolução do vídeo e da profundidade de cores no Servidor de Vídeo Adaptativo.

Referências Bibliográficas

[1] F. Adelstein, S. K. S. Gupta, G. G. R. III, and L. Schwiebert. *Fundamentals of Mobile and Pervasive Computing*. McGraw-Hill Professional Engineering, 2005.

[2] S. Agarwala, C. Poellabauer, J. Kong, K. Schwan, and M. Wolf. System-level resource monitoring in high-performance computing environments. *J. Grid Computing*, 1(3):273–289, 2003.

[3] D. E. Bakken. *Middleware*. Kluwer Academic Press, 2001.

[4] J. Boner. Aspectwerkz - dynamic aop for java.

[5] Borland. Janeva: A framework for interoperability of .net, j2ee and corba. URL: http://www.borland.de/janeva/index.html, 2006.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern - Oriented Software Architecture: A System of Patterns*, volume 1. John Wiley and Sons, 1996.

[7] F. J. da Silva e Silva. *Adaptação Dinâmica de Sistemas Distribuídos*. PhD thesis, Instituto de Matemática e Estatística da Universidade de São Paulo, Janeiro 2003.

[8] F. J. da Silva e Silva, F. Kon, J. Yoder, and R. Johnson. A pattern language for adaptive distributed systems. In *SugarLoafPLoP*, 2005.

[9] R. Y. de Camargo, A. Goldchleger, F. Kon, and A. Goldman. Checkpointing-based rollback recovery for parallel applications on the integrade grid middleware. In *MGC '04: Proceedings of the 2nd workshop on Middleware for grid computing*, pages 35–40, New York, NY, USA, 2004. ACM Press.

[10] R. Y. de Camargo, F. Kon, and R. Cerqueira. Strategies for checkpoint storage on opportunistic grids. *IEEE Distributed Systems Online*, 7(9):1, 2006.

[11] D. de Roure, M. Baker, N. Jennings, and N. Shadbolt. The evolution of the grid, 2003.

[12] S. A. de Sousa and F. J. da Silva e Silva. Flexible fault-tolerance of bsp applications on integrade grid middleware. Regular paper, Universidade Federal do Maranhão (UFMA), 2007.

[13] F. Dong and S. G. Akl. Scheduling algorithms for grid computing: State of the art and open problems. Technical Report 2006-504, School of Computing, Queens University, Kingston, Ontario, January 2006.

[14] H. A. Duran-Limon, G. S. Blair, and G. Coulson. Adaptive resource management in middleware: A survey. *IEEE Distributed Systems Online*, 05(7), 2004.

[15] T. Edmonds, A. Hopper, and S. Hodges. Pervasive adaptation for mobile computing. In *ICOIN '01: Proceedings of the The 15th International Conference on Information Networking*, page 111, Washington, DC, USA, 2001. IEEE Computer Society.

[16] C. Efstratiou and K. Cheverst. Reflection: A solution for highly adaptive mobile systems. In *Position paper at the Reflective Middleware Workshop in conjunction with Middleware 2000*, April 2000.

[17] J. Ferber. Computational reflection in class based object-oriented languages. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 317–326, New York, NY, USA, 1989. ACM.

[18] I. Foster. The anatomy of the grid: Enabling scalable virtual organizations. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 6, Washington, DC, USA, 2001. IEEE Computer Society.

[19] A. Frei, A. Popovici, and G. Alonso. Eventizing applications in an adaptive middleware platform. *IEEE Distributed Systems Online*, 6(4):1, 2005.

[20] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G. C. Bezerra. Integrade: Object-oriented grid middleware leveraging idle computing power of desktop machines. *Concurrency and Computation: Practice and Experience*, 16(5):449–459, March 2004.

[21] G. T. Heineman and W. T. Councill. *Component-Based Software Engineering: Putting the PiecesTogether*. Pearson Education, 2001.

[22] J. Hillman and I. Warren. Meta-adaptation in autonomic systems. In *FTDCS '04: Proceedings of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS'04)*, pages 292–298, Washington, DC, USA, 2004. IEEE Computer Society.

[23] J. Hillman and I. Warren. An open framework for dynamic reconfiguration. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 594–603, Washington, DC, USA, 2004. IEEE Computer Society.

[24] S. Hwang and C. Kesselman. Gridworkflow: A flexible failure handling framework for the grid. In *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, page 126, Washington, DC, USA, 2003. IEEE Computer Society.

[25] J. Kaufman, T. Lehman, G. Deen, and J. Thomas. Optimalgrid: Autonomic computing on the grid. URL: http://www-128.ibm.com/developerworks/library/gr-opgrid/.

[26] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The art of metaobject protocol.* MIT Press, Cambridge, MA, USA, 1991.

[27] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP: European Conference on Object-Oriented Programming*, 1997.

[28] F. Kon. *Automatic Configuration of Component-Based Distributed Systems.* PhD thesis, University of Illinois, Urbana, Illinois, 2000.

[29] R. Laddaga and P. Robertson. Self adaptive software: A position paper. In *SELF-STAR: International Workshop on Self-* Properties in Complex Information Systems*, Bertinoro, Italy, May 2004.

[30] H. Liu. *Accord: A Programming System for Autonomic Self-Managing Applications.* PhD thesis, Rutgers, The State University of New Jersey, October 2005.

[31] H. Liu and M. Parashar. Accord: A programming framework for autonomic applications. *Systems, Man and Cybernetics, Part C, IEEE Transactions on*, 36(3):341–352, May 2006.

[32] B. Lowekamp, N. Miller, R. Karrer, T. R. Gross, and P. Steenkiste. Design, implementation, and evaluation of the remos network monitoring system. *J. Grid Comput.*, 1(1):75–93, 2003.

[33] P. Maes. Concepts and experiments in computational reflection. In A. Press, editor, *Proceedings of Object-Oriented Programming Systems, Languages, and Applications Conference '87*, volume 22 of *Special Issue of Sigplan Notices*, pages 147–155. ACM, December 1987.

[34] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *HCW '99: Proceedings of the Eighth Heterogeneous Computing Workshop*, page 30, Washington, DC, USA, 1999. IEEE Computer Society.

[35] P. McKinley, S. Sadjadi, E. Kasten, and B. Cheng. A taxonomy of compositional adaptation. Technical Report MSU-CSE-04-17, Software Engineering and Network Systems Laboratory, Michigan State University, East Lansing, Michigan, July 2004.

[36] D. Moreto. Monitoramento de eventos compostos em sistemas distribuídos. Master's thesis, Instituto de Matemática e Estatística, Universidade de São Paulo (USP), Setembro 1998.

[37] A. Mukhija and M. Glinz. Casa a contract-based adaptive software architecture framework. In *Proceedings of the 3rd IEEE Workshop on Applications and Services in Wireless Networks (ASWN 2003)*, pages 275–286, Berne, Switzerland, 2003. IEEE Computer Society.

[38] A. Mukhija and M. Glinz. A framework for dynamically adaptive applications in a self-organized mobile network environment. In *ICDCSW '04: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04)*, pages 368–374, Washington, DC, USA, 2004. IEEE Computer Society.

[39] A. Mukhija and M. Glinz. The casa approach to autonomic applications. In *Proceedings of the 5th IEEE Workshop on Applications and Services in Wireless Networks (ASWN 2005)*, Paris, France, 2005. IEEE Computer Society.

[40] A. Mukhija and M. Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *ARCS*, pages 124–138, 2005.

[41] A. Nicoara and G. Alonso. Dynamic aop with prose. In *CAiSE Workshops (2)*, pages 125–138, 2005.

[42] Object Management Group (OMG). *Event Service Specification, version 1.2*, October 2004.

[43] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[44] M. Parashar, Z. Li, H. Liu, V. Matossian, and C. Schmidt. Enabling autonomic grid applications: Requirements, models and infrastructure. In *Self-star Properties in Complex Information Systems*, pages 273–290, 2005.

[45] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri. Automate: Enabling autonomic applications on the grid. *Cluster Computing*, 9(2):161–174, 2006.

[46] R. Rao. From the broad notion of reflection to the engineering practice of object-oriented metalevel architectures. In *Proceedings of the OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, 1991.

[47] A. Rasche and A. Polze. Configuration and dynamic reconfiguration of component-based applications with microsoft .net. In *ISORC '03: Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, page 164, Washington, DC, USA, 2003. IEEE Computer Society.

[48] A. Rasche and A. Polze. Dynamic reconfiguration of component-based real-time software. In *WORDS '05: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 347–354, Washington, DC, USA, 2005. IEEE Computer Society.

[49] A. Rasche, M. Puhlmann, and A. Polze. Heterogeneous adaptive component-based applications with adaptive.net. In *ISORC '05: Proceedings of the Eighth*

IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05), pages 418–425, Washington, DC, USA, 2005. IEEE Computer Society.

[50] A. Rashid and G. Kortuem. Adaptation as an aspect in pervasive computing. In *Building Software for Pervasive Computing at OOPSLA '04*, 2004.

[51] P. Rigole and Y. Berbers. The working of the seescoa common test case. Report CW 354, Department of Computer Science, K.U.Leuven, Leuven, Belgium, January 2003.

[52] P. Robertson, H. Shrobe, and R. Laddaga. *Self-Adaptive Software*, volume 1936 of *Lecture Notes in Computer Science*. First International Workhop, IWSAS 2000, Oxford, UK, 2000.

[53] M. A. S. Sallem and F. J. da Silva e Silva. Adapta: a framework for dynamic reconfiguration of distributed applications. In *ARM '06: Proceedings of the 5th workshop on Adaptive and reflective middleware (ARM '06)*, page 10, New York, NY, USA, 2006. ACM.

[54] M. A. S. Sallem and F. J. da Silva e Silva. The adapta framework for building self-adaptive distributed applications. In *ICAS '07: Proceedings of the Third International Conference on Autonomic and Autonomous Systems*, page 46, Washington, DC, USA, 2007. IEEE Computer Society.

[55] M. A. S. Sallem, S. A. de Sousa, and F. J. da Silva e Silva. Autogrid: Towards an autonomic grid middleware. *WETICE 2007: Proceedings of the 16th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 0:223–228, 2007.

[56] M. Satyanarayanan. Mobile computing: where's the tofu? *SIGMOBILE Mob. Comput. Commun. Rev.*, 1(1):17–21, 1997.

[57] R. E. Schantz and D. C. Schmidt. Middleware for distributed systems: Evolving the common structure for network-centric applications. In *Encyclopedia of Software Engineering*. Wiley and Sons, 2001.

[58] H. Schulzrinne, A. Rao, and R. Lanphier. Real-time streaming protocolo (rtsp), 1998.

[59] C. Souza and C. Maziero. Intercessão em tempo de implantação - uma abordagem reflexiva para a plataforma j2ee. In *XV Simpósio Brasileiro de Engenharia de Software*, pages 286–301, Rio de Janeiro, RJ, 2001. Anais do XV Simpósio Brasileiro de Engenharia de Software.

[60] T. M. Tom, J. Buckley, M. Zenger, and A. Rashid. Towards a taxonomy of software evolution. In *International Workshop on Unanticipated Software Evolution*, 2003.

[61] Y. Vandewoude and Y. Berbers. Component state mapping for runtime evolution. In *In Proceedings of the 2005 International Conference on Programming Languages and Compilers*, pages 230–236, Las Vegas, Nevada, USA, June 2005.

[62] Y. Vandewoude, P. Rigole, and D. Urting. Draco: An adaptive run-time environment for components. Report CW 372, Department of Computer Science, K. U. Leuven, Leuven, Belgium, December 2003.

[63] R. Vanegas, J. Zinky, D. Karr, J. Loyall, R. Schantzand, and D. Bakken. Quos run-time support for quality of service in distributed objects. In *in Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 98)*, pages 207–223, England, September, 1998.

[64] R. J. Walker, E. L. A. Baniassad, and G. C. Murphy. An initial assessment of aspect-oriented programming. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 120–130, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[65] I. Warren, J. Sun, S. Krishnamohan, and T. Weerasinghe. An automated formal approach to managing dynamic reconfiguration. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 37–46, Washington, DC, USA, 2006. IEEE Computer Society.

[66] J. W. Yoder and R. E. Johnson. The adaptive object-model architectural style. In *WICSA 3: Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*, pages 3–27, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.