UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

LUCAS LUIZ PROVENSI

# Uma Plataforma de Middleware Reflexivo com Suporte para Auto-Adaptação

Goiânia
2009

LUCAS LUIZ PROVENSI

# Uma Plataforma de Middleware Reflexivo com Suporte para Auto-Adaptação

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

**Área de concentração:** Ciência da Computação.

**Orientador:** Prof. Fábio Moreira Costa
**Co-Orientador:** Prof. Vagner José do Sacramento Rodrigues

Goiânia
2009

LUCAS LUIZ PROVENSI

# Uma Plataforma de Middleware Reflexivo com Suporte para Auto-Adaptação

Dissertação defendida no Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás como requisito parcial para obtenção do título de Mestre em Ciência da Computação, aprovada em 6 de Abril de 2009, pela Banca Examinadora constituída pelos professores:

---

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

---

**Prof. Vagner José do Sacramento Rodrigues**
Instituto de Informática – UFG

---

**Prof. Eduardo Simões de Albuquerque**
Instituto de Informática – UFG

---

**Prof. Ronaldo Alves Ferreira**
Departamento de Computação e Estatística – UFMS

**Lucas Luiz Provensi**

Graduou-se em Ciências da Computação pela Universidade Federal de Goiás. Durante a graduação, foi pesquisador do CNPq em um trabalho de iniciação científica envolvendo o estudo e implementação da plataforma de middleware MetaORB4Java. Durante o Mestrado, continuou seu trabalho com plataformas de middleware desenvolvendo a plataforma MetaORB.NET.

Este trabalho é dedicado aos meus pais, Gilmar e Sueli, e a minha amada Lorena, pelo amor incondicional, pelo apoio e por estarem sempre presentes.

# Agradecimentos

Gostaria de agradecer primeiramente ao meu orientador, Professor Fábio Moreira Costa, pela amizade, ajuda, paciência e, principalmente, pela confiança depositada em mim durante todos os anos nos quais trabalhamos juntos. Ao Professor Vagner Sacramento pelas inestimáveis contribuições neste trabalho, amizade e otimismo que sempre me deram motivação para continuar adiante. A Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo aporte financeiro.

Agradeço a todos os professores do Instituto de Informática pela ajuda dentro e fora da sala de aula, especialmente ao Professor Humberto José Longo pela ajuda neste trabalho e pelas conversas sempre proveitosas. A todos os funcionários do Instituto de Informática, principalmente a Edir Borges por sempre estar disposto a ajudar.

A todos os meus colegas pela amizade nestes últimos anos. Gostaria de agradecer em especial a Daniel Alves, Fernando Santos, Luciana Nishi e Raphael Gomes pelo companheirismo que tornaram os últimos anos mais divertidos.

Agradeço a todos os meus amigos e parentes pela preocupação e ajuda durante a graduação e o mestrado. Especialmente aos meus irmãos, Mateus e Laryssa, a Adriana Rosset e a Roseli Maestri pelo carinho, fé e prestatividade.

Por fim, agradeço a Deus por permitir que eu completasse mais uma etapa da minha vida com saúde, sem privações e por me cercar de amigos valiosos.

Um raciocínio lógico leva você de A a B. A imaginação leva você a qualquer lugar.

**Albert Einstein**.

# Resumo

Provensi, Lucas Luiz. **Uma Plataforma de Middleware Reflexivo com Suporte para Auto-Adaptação**. Goiânia, 2009. 116p. Dissertação de Mestrado. Instituto de Informática, Universidade Federal de Goiás.

O ambiente de computação distribuída atual é caracterizado pela crescente heterogeneidade, pelo dinamismo e por aplicações cada vez mais complexas. Aplicações para esse tipo de ambiente precisam de autonomia para realizar atividades de gerenciamento, tais como instalação, configuração e manutenção, com o propósito de diminuir o custo e o esforço envolvidos em tais atividades. Em ambientes móveis, por exemplo, as aplicações precisam se adaptar em função de mudanças no ambiente (largura de banda disponível, nível do sinal, etc.), que podem afetar a qualidade do serviço oferecida aos usuários. Este trabalho apresenta uma proposta para auto-adaptação baseada na arquitetura de middleware reflexivo Meta-ORB. O foco principal da proposta é prover suporte à auto-adaptação para aplicações que apresentam requisitos de qualidade de serviço. Neste trabalho, o conceito de políticas de adaptação foi introduzido no meta-modelo do middleware, permitindo que o desenvolvedor defina tanto o modelo das aplicações quanto seu comportamento adaptativo usando a mesma linguagem de modelagem. Esse modelo unificado fica disponível em tempo de execução, sendo usado por um mecanismo de auto-adaptação incorporado ao middleware. A arquitetura proposta foi implementada em um protótipo chamado MetaORB.NET, empregado em um estudo de caso para avaliar o uso da arquitetura no desenvolvimento de aplicações auto-adaptativas.

**Palavras–chave**

Middleware Reflexivo, Auto-adaptação, Meta-modelagem

# Abstract

Provensi, Lucas Luiz. **A Reflective Middleware Platform with Support to Self-Adaptation**. Goiânia, 2009. 116p. MSc. Dissertation. Instituto de Informática, Universidade Federal de Goiás.

Today's distributed computing environment is characterized by its increasing heterogeneity and dynamism and as well as the complexity of applications. Applications for this environment need autonomy to perform management activities, such as installation, configuration and maintenance, in order to reduce the cost and effort involved in such activities. In mobile environments, for example, applications need to adapt themselves according to changes in the environment (available bandwidth, signal level, etc.) that may affect the quality of service offered to users. This work presents an approach for self-adaptation based on the Meta-ORB reflective middleware architecture. The main focus of this approach is to provide support for self-adaptation to applications that have quality of service requirements. In this work, the concept of adaptation policy was introduced in the middleware meta-model, allowing the developer to define both the model of the applications and its adaptive behavior using the same modeling language. This unified model is available at runtime, being used by a self-adaptation mechanism incorporated in the middleware. The proposed architecture was implemented in a prototype called MetaORB.NET, used in a case study to evaluate the use of the architecture in the development of self-adaptive applications.

**Keywords**

Reflective Middleware, Self-Adaptation, Meta-modeling

# Sumário

# Lista de Figuras

# Lista de Tabelas

# Lista de Códigos de Programas

CAPÍTULO 1

# Introdução

No início do século 21, um dos maiores obstáculos da indústria de tecnologia da informação era a crescente complexidade das soluções de software. Alguns sistemas e aplicações continham milhões de linhas de código, o que dificultava ainda mais seu gerenciamento (instalação, configuração, manutenção, etc.) em cenários reais de produção. Para diminuir o custo e o esforço envolvidos no gerenciamento de tais aplicações surgiu o conceito de computação autonômica [33].

O paradigma de computação autonômica é baseado no sistema nervoso humano. O sistema nervoso pode identificar mudanças no ambiente por meio dos sentidos e comandar as ações que serão realizadas pelo corpo humano para entrar em equilíbrio com o ambiente, garantindo assim a sobrevivência do indivíduo frente a situações adversas. Em um sistema computacional autonômico a sobrevivência corresponde à habilidade de se reconfigurar automaticamente, de acordo com mudanças no ambiente, para se recuperar de falhas, manter o desempenho optimizado, etc [46].

Computação autonômica implica no desenvolvimento de sistemas autogerenciáveis (*self-management systems*) ou auto-adaptativos (*self-adaptive systems*). Esses sistemas possuem autonomia para, por exemplo, fazer o *download* e instalar atualizações, identificar um novo hardware conectado ao computador e instalar seu *driver*, etc. Idealmente, esses sistemas apresentam as seguintes características [30]:

1. **Autoconfiguração** (*Self-Configuration*): Inclusão de novas funcionalidades dinamicamente, sem a necessidade de parar a execução do sistema;
2. **Auto-Otimização** (*Self-Optimization*): O sistema busca optimizar seu desempenho e eficiência sempre que possível.
3. **Autocura** (*Self-Healing*): O sistema detecta, diagnostica e repara automaticamente erros de hardware e software.
4. **Autoproteção** (*Self-Protection*): O sistema se defende automaticamente de ataques maliciosos e falhas em cascata.

As características de auto-adaptativos são especialmente importantes para os sistemas distribuídos atuais, devido à sua crescente complexidade e por fazerem parte de

ambientes cada vez mais dinâmicos e heterogêneos. Um exemplo são os sistemas críticos, que não podem parar sua execução para efetuar manutenções e atualizações. Um outro exemplo são os ambientes de computação móvel, onde a comunicação é intermitente e os dispositivos possuem recursos limitados. Devido à flexibilidade de mobilidade, as aplicações precisam de autonomia para se adaptar a variações no ambiente, sem que isso interrompa sua execução. Aplicações auto-adaptativas para esse ambiente podem descobrir e instalar novos serviços, trocar o protocolo de comunicação utilizado por outro mais seguro etc.

No ambiente móvel, a auto-adaptação é bastante explorada nas aplicações sensíveis ao contexto. Essas aplicações podem se adaptar de acordo com informações de contexto extraídas do ambiente e dos dispositivos, tais como a localização do usuário, nível de bateria do dispositivo, conectividade, luminosidade do ambiente, etc. Um exemplo são as aplicações multimídia para ambientes móveis, que podem trocar dinamicamente o *codec* de compressão de áudio e vídeo de acordo com a largura de banda disponível.

As aplicações colaborativas para ambientes móveis são outro tipo de aplicação que pode se beneficiar da auto-adaptação, principalmente da auto-adaptação baseada em informações de contexto. Essas aplicações podem usar, por exemplo, informações sobre a localização do usuário e dos dispositivos próximos para criar grupos colaborativos dinamicamente. Além disso, algumas dessas aplicações envolvem interação em tempo real baseada em conteúdo multimídia, como áudio, vídeo e, mais recentemente, a tinta digital. Esse tipo de interação é sensível a variações dinâmicas no ambiente, que podem afetar a qualidade de serviço oferecida aos usuários, o que indica a necessidade de auto-adaptação.

O surgimento de novas tecnologias, como a própria tinta-digital, tende a aumentar ainda mais a heterogeneidade em ambientes computacionais. Com isso, cresce a necessidade de adequação das aplicações a mudanças em seu ambiente, que passa a conter novos tipos de dispositivos, recursos e serviços. Isso implica em novas funcionalidades, que podem ser inseridas por meio de adaptações de granularidade fina dos componentes que constituem a aplicação.

## 1.1 Justificativa

Apesar das características de auto-adaptação serem importantes em ambientes distribuídos, sua implementação como parte das aplicações pode aumentar consideravelmente a complexidade no desenvolvimento das mesmas. Desta forma, a implementação dessas características é mais adequada a uma camada de software isolada e reutilizável, tal como o *middleware* subjacente. Isso simplifica o desenvolvimento de aplicações auto-adaptativas, uma vez que o middleware pode oferecer o mecanismo adaptativo, cabendo

ao desenvolvedor definir apenas as políticas que determinam quando e como uma adaptação deve ser disparada.

Porém, a característica monolítica das primeiras plataformas de middleware, tais como Java RMI [60] e Microsoft DCOM [31], impossibilita a auto-adaptação, uma vez que o middleware não é capaz de acessar ou modificar sua própria estrutura interna ou a das aplicações. Assim, o principal pré-requisito para a auto-adaptação de middleware é que a plataforma seja "aberta", ou seja, capaz de inspecionar e adaptar sua estrutura interna e comportamento em tempo de execução [7]. Já plataformas de middleware abertas bem estabelecidas, como Open ORB [8] e DynamicTAO [35], oferecem a flexibilidade necessária para adaptação dinâmica por meio de reflexão computacional [38], embora não definam mecanismos para realizar adaptações de forma automática.

Diversos trabalhos estendem a abordagem típica de middleware adaptativo e reflexivo inserindo mecanismos para auto-adaptação, como em [28] e [32], que serão discutidos com maiores detalhes no Capítulo 6. Em comum, esses trabalhos não consideram o gerenciamento das políticas de adaptação, e utilizam linguagens complexas para defini-las e bancos de dados externos para armazená-las. Essa abordagem dificulta significativamente o desenvolvimento de aplicações auto-adaptativas em cenários reais de produção. Uma solução de middleware mais adequada para o desenvolvimento deste tipo de aplicação não deve comprometer as outras funcionalidades do middleware ou aumentar a complexidade no desenvolvimento das aplicações. Além disso, a infra-estrutura que implementa a auto-adaptação não deve sobrecarregar o sistema, aumentando ainda mais o consumo de memória, de processador, de largura de banda, etc.

## 1.2 Objetivos

Tendo em vista a discussão apresentada na Seção 1.1, os principais objetivos deste trabalho são:

- Propor um arquitetura de middleware reflexivo e auto-adaptativo para atender as necessidades das aplicações distribuídas, em especial as que apresentam requisitos de qualidade de serviço;
- Facilitar o gerenciamento de políticas de adaptação e, conseqüentemente, o desenvolvimento de aplicações auto-adaptativas;
- Desenvolver um protótipo funcional da arquitetura para demonstrar sua viabilidade;
- Implementar uma aplicação de estudo de caso para avaliar o impacto da utilização da arquitetura em um cenário real.

## 1.3 Metodologia

A arquitetura auto-adaptativa proposta nesta dissertação foi construída como uma extensão da arquitetura de middleware reflexivo Meta-ORB [17]. A abordagem Meta-ORB já foi explorada em trabalhos anteriores que contaram com a participação do autor, tais como [48] e [18]. O primeiro, consistiu na implementação de um protótipo da plataforma Meta-ORB em linguagem Java, chamado MetaORB4Java. O segundo consistiu no projeto de um sistema para o gerenciamento da meta-informação usada pelo middleware, baseado na tecnologia de meta-modelagem EMF (*Eclipse Modeling Framework*) [10].

Para o trabalho atual, um novo protótipo da arquitetura Meta-ORB foi desenvolvido, desta vez incluindo extensões de suporte a auto-adaptação. O novo protótipo, chamado MetaORB.NET, foi escrito em linguagem C#, na plataforma .NET. O motivo do desenvolvimento de um novo protótipo, e não a utilização protótipo MetaORB4Java, é o aumento do domínio das aplicações que podem ser desenvolvidas com a ajuda da plataforma. Isso inclui aplicações como a do estudo de caso, apresentada no Capítulo 5, que utiliza a tecnologia de tinta digital através da biblioteca Microsoft.Ink [65].

Na abordagem proposta nesta dissertação, as políticas de adaptação são usadas pelo middleware como meta-informação, definindo quando e como as adaptações devem ser realizadas. Desta forma, o meta-modelo do middleware foi estendido para suportar o conceito de políticas de adaptação. Como o sistema de gerenciamento de meta-informação é baseado em uma tecnologia implementada em Java (EMF), um serviço remoto para acessá-lo foi desenvolvido e integrado ao protótipo MetaORB.NET.

Apesar da possibilidade do uso do middleware em diferentes domínios de aplicações distribuídas, as aplicações colaborativas baseadas em tinta digital foram escolhidas como estudo de caso. Essas aplicações utilizam uma tecnologia relativamente nova, cujos requisitos para comunicação em ambientes móveis ainda são pouco estudados. Para levantar os requisitos desse tipo de aplicação, foi realizada uma série de experimentos envolvendo sua usabilidade em um ambiente móvel, os quais serão detalhados no Capítulo 5. Uma vez identificados os requisitos, uma aplicação auto-adaptativa que consiste em um quadro branco compartilhado foi desenvolvida para avaliar a arquitetura e sua aplicabilidade.

## 1.4 Contribuições

De maneira geral, considerando os objetivos apresentados na Seção 1.2, as principais contribuições deste trabalho podem ser sumarizadas nos seguintes pontos:

- Aplicação de meta-modelagem de middleware na definição de políticas de adaptação, o que representa uma nova abordagem para a descrição de políticas de adaptação;
- Extensão da arquitetura Meta-ORB, tornando-a uma plataforma de middleware auto-adaptativo;
- Desenvolvimento de um protótipo da plataforma Meta-ORB, escrito em C#, expandindo assim sua aplicabilidade a outros domínios de aplicações distribuídas (por exemplo, aplicações baseadas em tinta digital);
- Estudo preliminar dos requisitos para a comunicação de tinta digital e de sua caraterização como um tipo de mídia de tempo-real, indicando a necessidade de um tratamento adequado por parte do middleware;
- Desenvolvimento de uma aplicação colaborativa auto-adaptativa baseada em tinta digital (quadro branco compartilhado), que pode ser aplicada em cenários reais (por exemplo, no ensino).

## 1.5 Estrutura da Dissertação

O restante da dissertação está estruturado da seguinte maneira. O Capítulo 2 apresenta uma discussão preliminar sobre a plataforma de middleware reflexivo Meta-ORB em termos dos princípios empregados em sua arquitetura, seu modelo de programação e seu mecanismo reflexivo de adaptação dinâmica.

O Capítulo 3 apresenta a arquitetura de auto-adaptação proposta nesta dissertação, discutindo a meta-modelagem de políticas de adaptação e o design dos mecanismos de monitoramento de contexto e gerenciamento de adaptação. O Capítulo 4 apresenta o protótipo MetaORB.NET, que representa a implementação concreta da arquitetura proposta, bem como sua integração com o sistema de gerenciamento de meta-informação da plataforma.

O Capítulo 5 apresenta o estudo de caso envolvendo o quadro branco compartilhado, discutindo os requisitos desta aplicação, os problemas envolvidos na sua utilização em um ambiente móvel, sua implementação como uma aplicação auto-adaptativa e uma avaliação da utilização da arquitetura neste cenário. O Capítulo 6 discute alguns trabalhos relacionados à proposta apresentada nesta dissertação. Por fim, o Capítulo 7 conclui a dissertação com uma discussão geral sobre suas principais contribuições e apresenta algumas propostas de trabalhos futuros.

CAPÍTULO 2

# A Plataforma Meta-ORB

A flexibilidade de configuração e reconfiguração das plataformas de middleware reflexivo as tornam ideais para o auto-gerenciamento de sistemas com requisitos dinâmicos, como aqueles apresentados por aplicações para o ambientes móveis e ubíquos, aplicações multimídia e de tempo-real, aplicações críticas, etc. Nesta dissertação, a proposta de middleware auto-adaptativo é baseada na plataforma de middleware Meta-ORB [15]. A arquitetura Meta-ORB combina técnicas de gerenciamento de meta-informação, que possibilitam a definição de configurações especializadas da plataforma, com reflexão computacional, que possibilita sua adaptação dinâmica.

Este capítulo apresenta uma revisão da arquitetura Meta-ORB e está estruturado como se segue. A Seção 2.1 discute brevemente os princípios empregados na construção da plataforma. A Seção 2.2 descreve os principais conceitos de seu modelo de programação, enquanto a Seção 2.3 apresenta o meta-modelo que define o modelo de programação da plataforma. A Seção 2.4 discute o mecanismo reflexivo de adaptação dinâmica. Por fim, a Seção 2.5 apresenta um breve histórico do projeto e a Seção 2.6 apresenta uma discussão sobre a inclusão de suporte para a auto-adaptação na arquitetura do middleware.

## 2.1 Princípios

Meta-informação, no escopo deste trabalho, descreve a estrutura e a semântica de entidades de um sistema. Essa descrição é utilizada para a configuração estática do middleware (instanciação dos componentes durante o carregamento do sistema) e para sua reconfiguração dinâmica (durante a execução do sistema). O gerenciamento de meta-informação é a base da abordagem de middleware configurável e adaptativo adotada pela plataforma Meta-ORB. Nessa abordagem, meta-modelagem e reflexão são técnicas centrais e utilizam o mesmo modelo de meta-informações. Essas técnicas serão discutidas a seguir.

### 2.1.1 Meta-modelagem

Modelagem implica em uma forma de representação de alto nível de objetos reais ou computacionais. Para isso, são consideradas apenas as propriedades comuns a todos os objetos pertencentes a um grupo ou domínio específico. Esse processo de abstração pode ser aplicado recursivamente, resultando em modelos que representam outros modelos ou meta-modelos, como são chamados [63].

Na modelagem de sistemas orientados a objetos, por exemplo, objetos com características similares são representados como classes e as conexões entre esses objetos como relacionamentos. Classes e relacionamentos são as construções que podem ser empregadas na modelagem dos objetos e, portanto, são conceitos representados no meta-modelo do modelo de objetos [40]. Os meta-modelos permitem a introspecção e instanciação de modelos, do mesmo modo que os modelos permitem a introspecção e instanciação de objetos.

A especificação MOF (*Meta-Object Facility*) [42] oferece uma arquitetura de meta-modelagem para o gerenciamento de meta-informação organizada em níveis. Cada nível da arquitetura é a instanciação do modelo do nível superior. Em teoria não existe um limite para o número de níveis. Na prática, a maioria dos sistemas utiliza até quatro níveis. Exemplificando com a linguagem UML (*Unified Modeling Language*) [43], a divisão em quatro níveis é feita da seguinte forma:

- **Nível 0** - Contém as entidades ou objetos que formam o sistema em tempo de execução;
- **Nível 1** - Contém os modelos de aplicações e sistemas específicos;
- **Nível 2** - Contém o meta-modelo de UML, que provê a notação utilizada para a modelagem de sistemas orientados a objetos;
- **Nível 3** - Contém o modelo MOF padrão, que é um meta-meta-modelo ou a linguagem utilizada para a definição de meta-modelos;

A divisão em quatro níveis também foi adotada na meta-modelagem da arquitetura Meta-ORB. Nesse contexto, o Nível 3 continua com o modelo MOF padrão. O Nível 2 passa a representar o sistema de tipos da plataforma, ou o meta-modelo utilizado para definir as construções que podem ser empregadas na modelagem do middleware e das aplicações. O Nível 1 é a definição das entidades que formam uma configuração particular do middleware, ou seja, o modelo utilizado para guiar a instanciação tanto da plataforma quanto das aplicações em tempo de execução.

A Figura 2.1 ilustra a meta-modelagem de componentes na arquitetura Meta-ORB. No Nível 0, os componentes que formam uma configuração de middleware em tempo de execução são instâncias de definições de componentes do Nível 1 (modelo),

que, por sua vez, são descritos de acordo com as construções que definem o que é um componente no Nível 2 (meta-modelo).

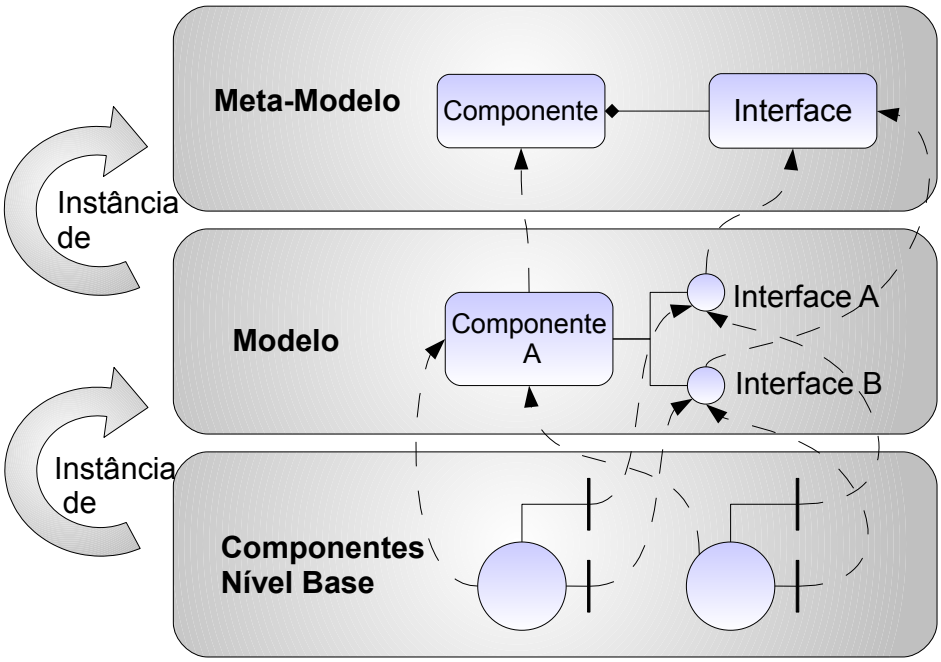



**Figura 2.1:** *Meta-modelam de componentes.*

De acordo com o processo de gerenciamento de modelos e meta-modelos da MOF, um meta-modelo pode ser definido com uma ferramenta de meta-modelagem, como a própria MOF, e compilado para gerar um repositório de modelos. O repositório oferece mecanismos para acesso, criação e modificação de modelos (tipos) construídos de acordo com o meta-modelo a partir do qual o repositório foi gerado.

O repositório de modelos, ou repositório de tipos, é essencial para a configurabilidade na arquitetura Meta-ORB e é baseada na estrutura do repositório de interfaces de CORBA [41]. Antes da execução do middleware, as definições dos tipos que formam sua configuração inicial são recuperadas a partir de um repositório acessível globalmente, sendo usadas como modelo para a instanciação das entidades do Nível 0, feita por fábricas especializadas. Em tempo de execução, os tipos continuam acessíveis no repositório, possibilitando a inspeção do modelo que define as entidades atualmente em execução, ou mesmo a instanciação de novas entidades.

### 2.1.2 Reflexão

Um sistema é dito *reflexivo* quando mantém uma auto-representação causalmente conectada com seu estado e comportamento, de modo que a manipulação dessa auto-representação seja refletida no sistema em si e vice versa [38]. Com o uso da auto-

representação, a estrutura interna e o comportamento de um sistema computacional podem ser inspecionados e adaptados.

Sistemas reflexivos são geralmente estruturados em níveis. No nível mais baixo, ou nível base, estão as entidades computacionais do domínio da aplicação, como objetos ou componentes em execução. Em um nível acima, ou meta-nível, estão as entidades computacionais para a representação e manipulação das entidades do nível inferior.

As entidades do meta-nível são implementadas com a mesma linguagem das entidades do nível base e geralmente são criadas sob demanda, no primeiro acesso a cada entidade do meta-nível. As entidades presentes no meta-nível também estão sujeitas a reflexão, o que resulta na criação de um meta-meta-nível, tendo o meta-nível como seu nível base. Esse processo pode continuar infinitamente, resultando em torres infinitas de meta-níveis, como ilustra a Figura 2.2.

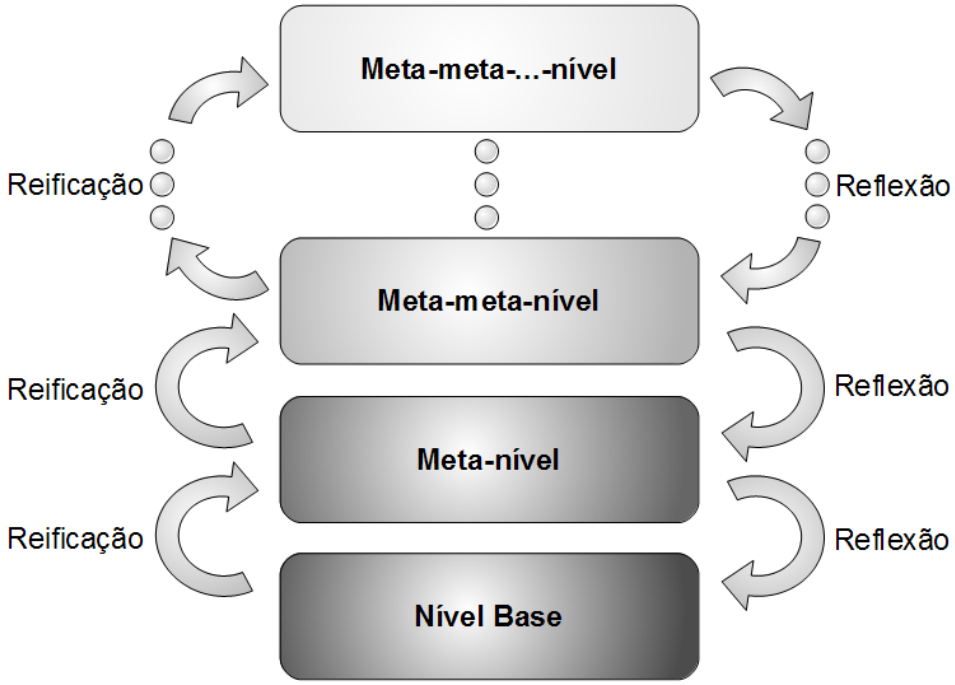



**Figura 2.2:** *Reflexão: Torre de meta-níveis.*

A Figura 2.2 ilustra ainda o conceito de *reificação*, que é a exposição da estrutura interna do nível base para construir sua auto-representação no meta-nível. A manipulação da auto-representação exposta pelo meta-nível resulta em mudanças no nível base, em um processo conhecido como *reflexão*.

O conceito de reflexão também pode ser aplicado às plataformas de middleware, resultando em plataformas de middleware reflexivo. Esse tipo de middleware expõe uma auto-representação causalmente conectada ao middleware em si, que permite a sua inspeção e adaptação em tempo de execução [19]. Com isso, o middleware torna-se flexível o suficiente para alterar sua estrutura e comportamento dinamicamente.

A plataforma Meta-ORB segue os princípios da arquitetura reflexiva de Open ORB [8]. Essa arquitetura é, da mesma forma, dividida em um nível base, contendo as funcionalidades do middleware e das aplicações, e um meta-nível, que provê a reificação do nível base. Tanto o nível base quanto o meta-nível são definidos em termos de um modelo de componentes uniforme, que será apresentado na Seção 2.2, tendo como base um meta-modelo explícito, apresentado na Seção 2.3. A arquitetura reflexiva da plataforma é apresentada na Seção 2.4.

## 2.2 Modelo de Programação

A plataforma Meta-ORB define um modelo de programação baseado em componentes de software. Componentes são unidades de software projetadas para composição, que possuem interfaces especificadas contratualmente e que contêm apenas dependências explícitas [61]. Componentes podem ser distribuídos independentemente e estão sujeitos à composição com outros componentes criados por terceiros. Um modelo de componentes deve promover a reutilização, uma vez que componentes podem participar de diversas composições com finalidades distintas, e também a configurabilidade, uma vez que componentes com interfaces compatíveis podem ser combinados para formar configurações tanto estática quanto dinamicamente.

Um sistema criado com o modelo de programação da plataforma Meta-ORB, é formado por *componentes*, suas *interfaces* e *objetos de binding* (ou simplesmente *bindings*). Os componentes encapsulam as funcionalidades específicas do sistema. As interfaces são os únicos pontos de acesso aos componentes e definem um ponto de contrato entre os componentes e seus usuários, em termos de serviços que o componente oferece e serviços que o componente utiliza. *Bindings*, por sua vez, são objetos distribuídos ao longo de um ou mais espaços de endereçamento e que têm o propósito de conectar as interfaces de componentes localizados remotamente entre si.

### 2.2.1 Componentes e Interfaces

A Figura 2.3 ilustra um componente e sua interface, utilizando a notação que será empregada no decorrer do texto para representar componentes e interfaces. O componente *RSACryptoComp*, exemplificado na Figura 2.3, encapsula uma implementação do algoritmo de criptografia RSA. A interface *CryptoInterface* oferece acesso às funcionalidades do componente através das operações *GenerateKeys*, *Encrypt* e *Decrypt*, que podem ser usadas respectivamente para gerar o par de chaves de criptografia, criptografar uma mensagem e descriptografar uma mensagem. Esse componente é classificado como

*primitivo*, pois encapsula o código de implementação de uma funcionalidade, dependente de linguagem de programação e de plataforma de execução.

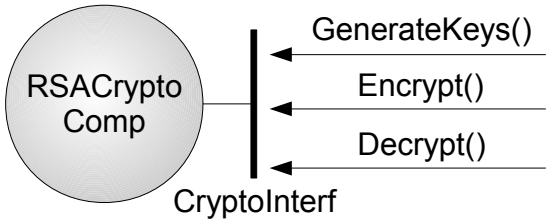



**Figura 2.3:** *Exemplo de um componente primitivo.*

Os componentes primitivos oferecem funcionalidades atômicas, que podem ser combinadas para formar um componente composto. Um componente composto é visto externamente como um único componente, apesar de internamente ser formados por outros componentes. Desta forma, a funcionalidade do componente composto é exposta por meio de uma ou mais interfaces de seus componentes internos, que passam a ser vistas como as interfaces do componente composto.

O componente primitivo *RSACryptoComp*, por exemplo, pode ser combinado com outros componentes para oferecer um sistema criptográfico mais completo. A Figura 2.4 ilustra esse sistema encapsulado no componente composto *CryptoSystem*. Na figura, além do componente *RSACryptoComp*, que define o algoritmo de criptografia usado, existe o componente *KeyManager*, para gerenciar as chaves usadas, e o componente *CryptographyManager*, que expõe as funcionalidades do sistema de criptografia. O usuário acessa o componente composto *CryptoSystem* através da interface *CryptoMainInterf*, que define operações para criptografar ou descriptografar uma mensagem. A interface *CryptoMainInterf* é uma das interface do componente interno *CryptographyManager*, que por sua vez está conectado aos componentes *KeyManager* e *RSACryptoComp*.

### 2.2.2 Bindings

Para completar o modelo de programação, existem os objetos de *binding*. Os *bindings* encapsulam vários aspectos da interação entre componentes, como a comunicação básica de rede, a serialização e deserialização de dados, etc. A plataforma Meta-ORB não define uma estrutura fixa de *binding*, e sim construções em seu meta-modelo que permitem a definição de *bindings* customizados.

Em tempo de execução, um *binding* consiste em uma infra-estrutura distribuída composta por componentes e outros *bindings*, caracterizando *bindings compostos*. No nível mais interno de um *binding* composto existe um *binding primitivo*, que encapsula o protocolo de transporte dependente de linguagem de programação e plataforma. A Figura 2.5 ilustra um *binding* entre componentes de um sistema de troca de mensagens

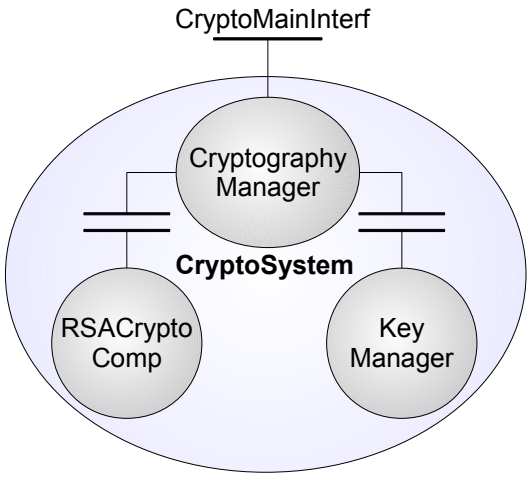



**Figura 2.4:** *Exemplo de um componente composto.*

(*Chat*), localizados em duas máquinas remotas (M1 e M2) interconectadas por uma rede. Em cada uma das máquinas, o *binding* é formado internamente por dois componentes (A e B) e por um *binding* primitivo (C).

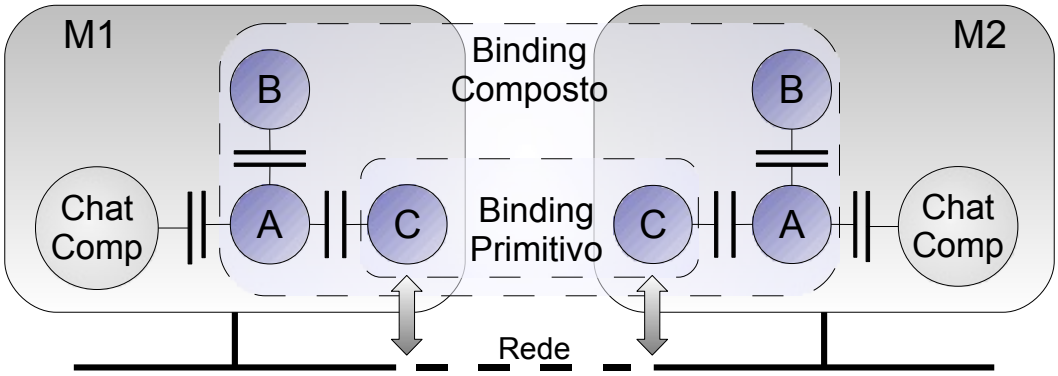



**Figura 2.5:** *Exemplo de um binding.*

Cada um dos componentes e *bindings* internos que formam um *binding* composto é responsável por um aspecto da interação entre os componentes remotos. No *binding* mostrado na Figura 2.5 por exemplo, o *binding* primitivo implementa o mecanismo de comunicação em rede, que pode ser baseado em algum protocolo de transporte como TCP ou UDP. O componente A realiza o papel de um *stub*, responsável por serializar as mensagens enviada pelo componente *ChatComp* ou deserializar as mensagens que chegam do *binding* primitivo. Já o componente B pode ser o componente *CryptoSystem* mostrado na Figura 2.4, usado pelo componente A para criptografar ou descriptografar as mensagens.

Os *bindings* compostos são conhecidos também como *bindings explícitos*, pois são criados explicitamente pelo programador resultando na infra-estrutura de comuni-

cação que liga as interfaces de componentes remotos [24]. A plataforma provê suporte também para outros dois tipos de *bindings*: *binding local* e *binding implícito*. Esses tipos de *binding*, no entanto, são artefatos primitivos do modelo de programação e não são descritos no meta-modelo ou sujeitos a configuração.

Os *bindings* locais são essenciais à configurabilidade, uma vez que implementam o ponto de interação entre duas interfaces localizadas em um mesmo espaço de endereçamento [9]. *Bindings* locais não possuem estrutura interna e consistem na conexão de interfaces compatíveis de componentes para formar uma configuração local. Duas interfaces são compatíveis quando os serviços oferecidos por uma são equivalentes aos serviços requeridos pela outra.

*Bindings* implícitos possibilitam a invocação direta de interfaces de componentes remotos, sem a necessidade de criação de uma estrutura de *binding* específica. Esse tipo de *binding* utiliza um mecanismo de comunicação primitivo da plataforma que não está sujeito a configuração. *Bindings* implícitos utilizam referências de interface, que também são artefatos primitivos do modelo de programação, como forma de localização de interfaces remotas. Uma referência de interface é um identificador estruturado, não-ambíguo e independente de localização de uma interface. Além de serem usadas para *bindings* implícitos, as referências de interfaces são utilizadas para identificar as interfaces que serão conectadas durante a criação de um *binding* explícito.

## 2.3 Meta-modelo

As principais construções do modelo de programação (componentes, interfaces e *bindings*) correspondem aos tipos de primeira classe contidos no repositório de tipos. Como visto na Seção 2.1.1, o repositório de tipos oferece acesso a modelos (tipos) criados a partir de um meta-modelo. Os tipos de primeira classe definem *templates* usados na instanciação das entidades que formam o sistema em tempo de execução e são empregados em conjunto com tipos auxiliares e outros tipos descritos no meta-modelo, para construir uma configuração particular de middleware. O restante da seção apresenta um resumo dos principais elementos do meta-modelo da plataforma Meta-ORB, cuja descrição completa vai além do escopo da dissertação e pode ser encontrada em [15].

A Figura 2.6 apresenta uma visão geral do meta-modelo da plataforma, organizado em pacotes. O pacote *baseIDL* contém as construções básicas, derivadas do meta-modelo de CORBA, que podem ser usadas na definição de modelos. O pacote *componentsAndBindings* contém as construções utilizadas na definição de componentes e objetos de *binding*. O pacote *interfaces* contém construções para a definição das interfaces dos componentes e, por fim, o pacote *mediaQoS* contém as construções utilizadas para associar as interfaces a tipos de mídia e a atributos de qualidade de serviço (*QoS - Quality of Service*).

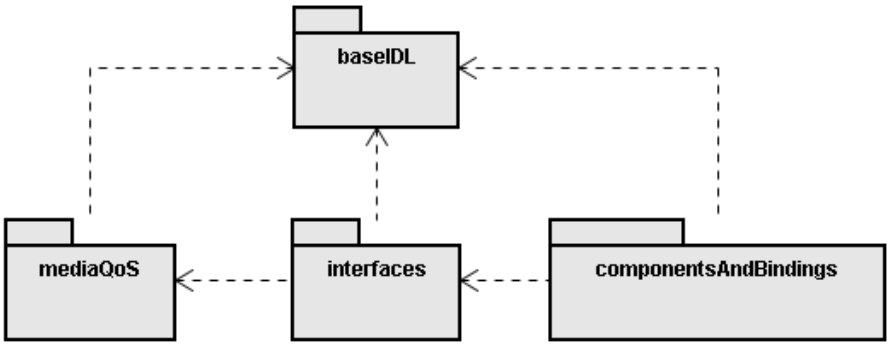



**Figura 2.6:** *Organização em pacotes dos elementos do meta-modelo da plataforma Meta-ORB.*

As principais construções presentes no pacote *baseIDL* são os meta-tipos *Contained* e *Container*, derivados do meta-modelo de CORBA, que são a base para a organização hierárquica dos tipos do meta-modelo. *Contained* é a super-classe para todas as entidades nomeadas da hierarquia, contendo atributos para a identificação da entidade no repositório de tipos, como nome, identificador único, versão, etc. Já *Container* é a super-classe para todas as entidades que podem conter outras entidades como parte de sua definição, formando assim uma hierarquia de definições. A Figura 2.7 ilustra a relação entre esses meta-tipos, que servem como base para as principais construções do meta-modelo, como componentes, interfaces e *bindings*. Para impedir que os meta-tipos derivados de *Container* possam conter qualquer meta-tipo derivado de *Contained*, são empregadas restrições no meta-modelo para definir quais meta-tipos eles podem efetivamente conter.

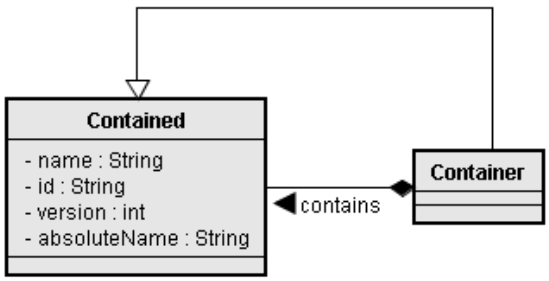



**Figura 2.7:** *Contained e Container.*

### 2.3.1 Interfaces

O meta-modelo da plataforma Meta-ORB define três estilos de interfaces:

- **Operacional** - Utilizada para interações do tipo cliente-servidor, onde a interface expõe as operações providas pelo componente, no caso de um servidor, ou as operações requeridas pelo componente, no caso de um cliente.

- **Sinal** - Utilizada para interações assíncronas entre um componente produtor e um consumidor de sinais atômicos. A interface expõe os sinais de saída (enviados pelo componente) e os sinais de entrada (recebidos pelo componente).
- **Fluxo Contínuo** - Utilizadas para interações de mídia contínua. Interfaces deste estilo podem expor fluxos de mídia de saída (enviados por um componente produtor) e fluxos de entrada (recebidos por um consumidor). Assim como as interfaces de sinal, os fluxos são enviados de maneira assíncrona, embora não sejam atômicos e sim seqüenciais.

A Figura 2.8 mostra o meta-modelo dos três estilos de interface. De acordo com o meta-modelo, uma definição de interface é um *Container* que pode ser especializada por cada um dos três estilos de interface. O conteúdo das interfaces pode ser formado por definições de operações, sinais ou fluxos de acordo com o estilo da interface. Além disso, as interfaces podem ser associadas a interfaces base, das quais herdam interações (operações, sinais e fluxos).

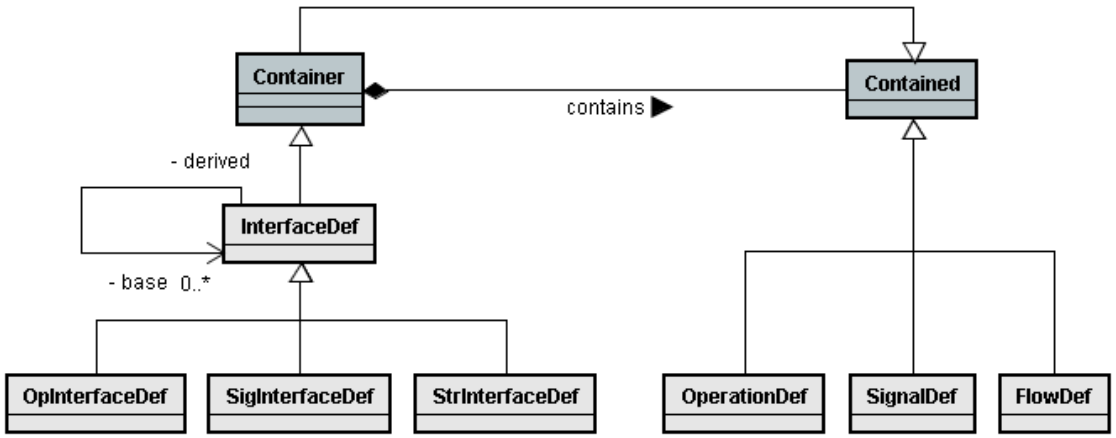



**Figura 2.8:** *Meta-modelagem dos estilos de interfaces no Meta-ORB.*

O meta-modelo permite ainda a definição de atributos de QoS para as interações contidas nas interfaces, bem como a associação de fluxos a tipos de mídia, como mostra de maneira simplificada a Figura 2.9. Cada operação, sinal ou fluxo é uma especialização do meta-tipo *QoSConstrained* e pode definir uma série de atributos de QoS que restringem a interação, como por exemplo o valor máximo tolerável para o atraso em uma interação de fluxo contínuo. Diferente dos sinais, associados a valores, e das operações, associadas a parâmetros, os fluxos são associados a especificações de mídia, que são *Containers* que contêm os tipos de mídia que podem ser usados na interação.

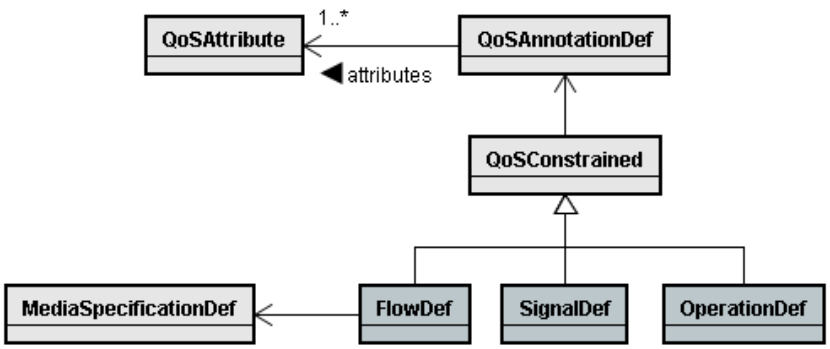



**Figura 2.9:** *Atributos de QoS e especificação de mídia.*

### 2.3.2 Componentes

O meta-modelo da plataforma provê suporte para dois tipos de componentes: *primitivos* e *compostos*, definidos na Seção 2.2.1. A Figura 2.10 ilustra o meta-modelo de componentes. Os componentes primitivos estão associados a um conjunto de interfaces através do meta-tipo *PrimInterface*, que expõe as interfaces desse tipo de componente. Os componentes compostos estão associados a um conjunto de componentes internos e definem um grafo de conexões entre esses componentes, formando sua configuração interna. Além disso, os componentes compostos estão associados a um conjunto de interfaces através do meta-tipo *CompInterface*, que define um mapeamento das interfaces dos componentes internos que são expostas.

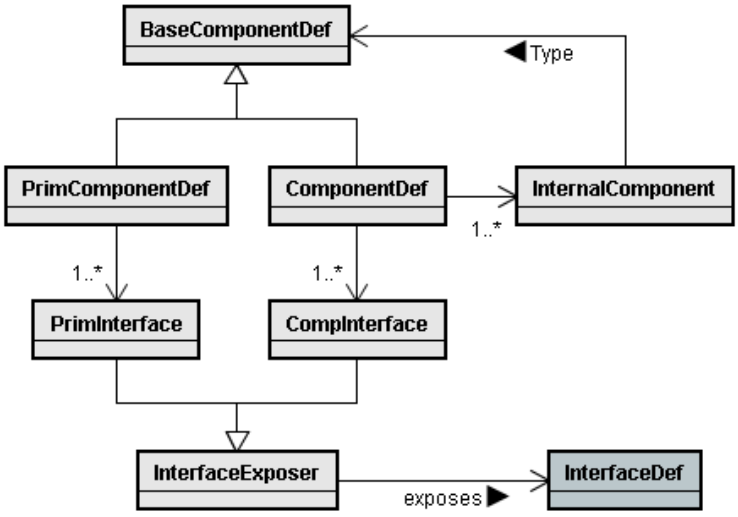



**Figura 2.10:** *Meta-modelagem de componentes primitivos e compostos.*

A Figura 2.11 ilustra a relação entre o meta-modelo mostrado na figura 2.10, o modelo de um componente criado a partir desse meta-modelo e as entidades que

formam o componente em tempo de execução. Essa relação corresponde aos níveis de meta-modelagem ilustrados anteriormente na Figura 2.1. O componente composto *CryptoSystem* e seus componentes internos são usados como exemplo. O modelo de um componente particular, ou seja, a definição do tipo desse componente, é criado de acordo com o meta-modelo da plataforma e armazenado no repositório de tipos. Em tempo de execução, o componente é criado por uma fábrica especializada (*ComponentFactory*), que acessa o repositório de tipos e obtém a definição do tipo do componente. Essa definição contém a meta-informação necessária para instanciar o componente.

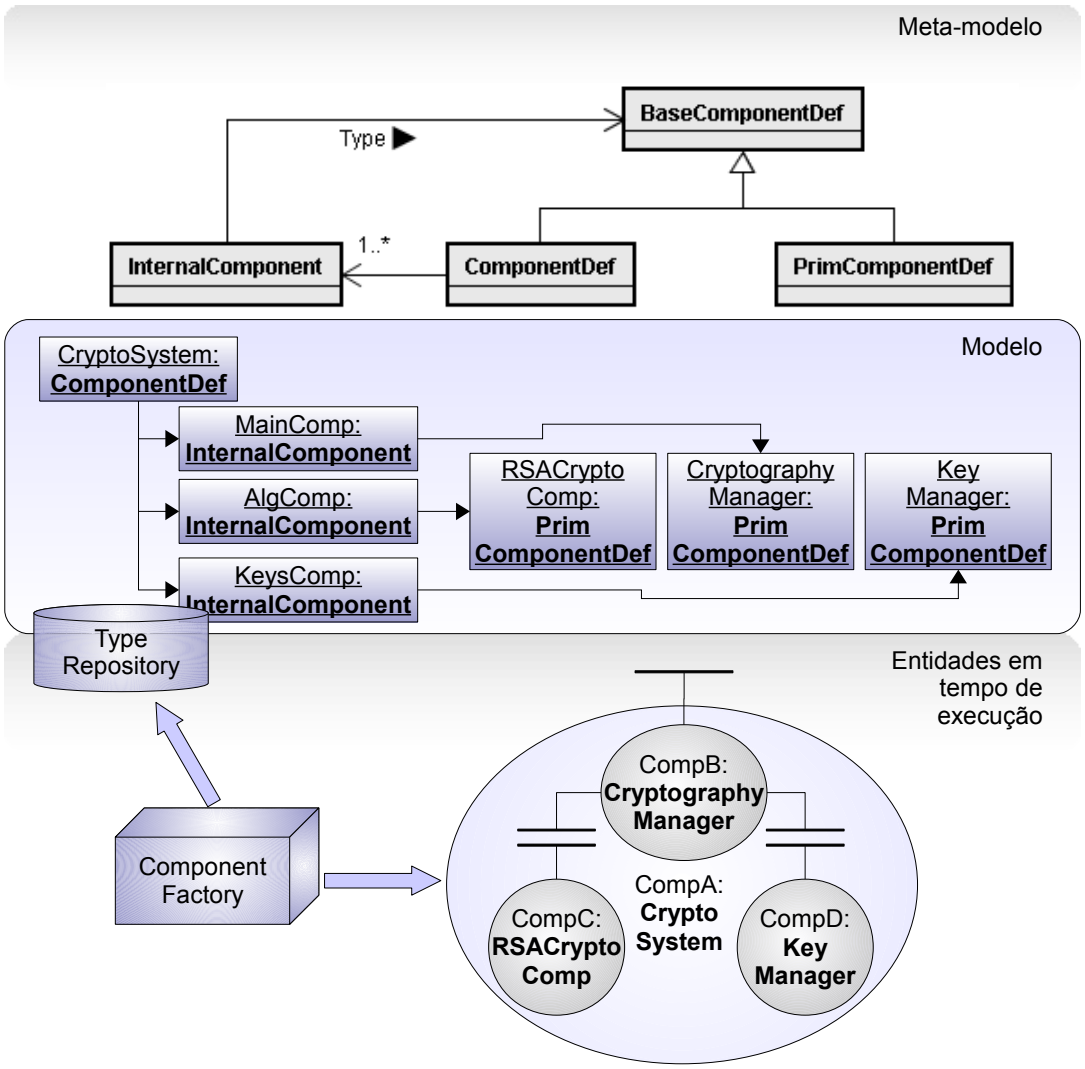



**Figura 2.11:** *Componentes: relação entre meta-modelo, modelo e entidades em tempo de execução.*

### 2.3.3 Binding

A Figura 2.12 mostra o meta-modelo de *bindings* na plataforma Meta-ORB. O meta-modelo permite a definição de dois tipos de *bindings*: *primitivos* e *compostos*, definidos na Seção 2.2.2. De acordo com o meta-modelo mostrado na Figura 2.12, *bindings* são *Containers* que podem conter uma ou mais definições de papéis (*Roles*). Os papéis definem a configuração interna do *binding* em um certo ponto de sua distribuição e a interface alvo do componente que será ligada ao *binding* nesse ponto. Cada ponto de distribuição, ou espaço de endereçamento, que faz parte de um *binding* é conhecido como *endpoint*.

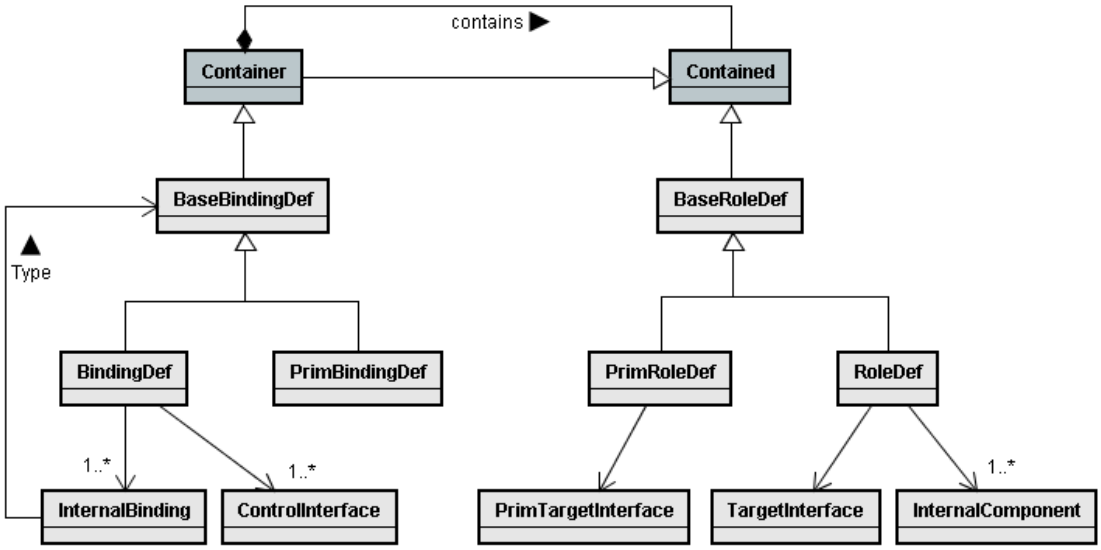



**Figura 2.12:** *Meta-modelagem de objetos de binding.*

Um mesmo *binding* pode definir diferentes papéis para diferentes tipos de interface que serão ligadas ao *binding*. Para *bindings* primitivos, os papéis definem apenas as interfaces alvo do *binding*. Para *bindings* compostos, os papéis definem tanto a interface alvo quanto uma configuração interna de componentes que realizam o processamento de vários aspectos da interação. Os *bindings* compostos possuem também um conjunto de interfaces de controle que possibilitam a interação com esses objetos, como se fossem componentes.

A Figura 2.13 ilustra a relação entre o meta-modelo mostrado na Figura 2.12, o modelo de um *binding* criado a partir desse meta-modelo e as entidades que formam o *binding* em tempo de execução. O exemplo mostrado na figura é o *binding* do sistema de troca de mensagens discutido na Seção 2.2. Os *bindings*, assim como os componentes, são criados por fábricas especializadas (*BindingFactory*) que usam definições obtidas no repositório de tipos como meta-informação para instanciar as entidades em tempo de execução. Entretanto, como o *binding* é um objeto distribuído entre vários pontos de

uma rede, o processo de criação é distribuído entre fábricas presentes em cada um dos extremos (*endpoints*), responsáveis por criar localmente cada parte da configuração do *binding*.

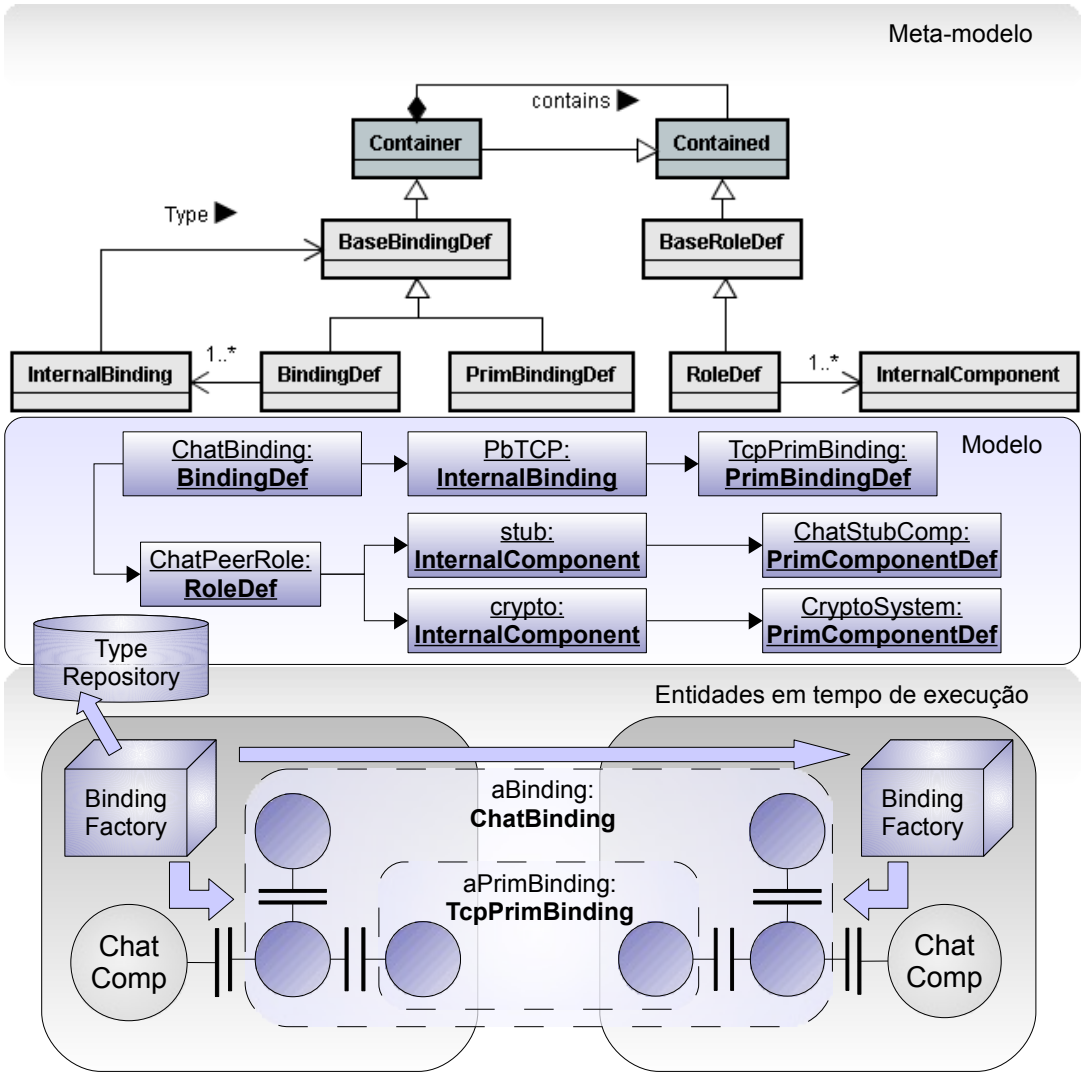



**Figura 2.13:** *Binding: relação entre meta-modelo, modelo e entidades em tempo de execução.*

## 2.4 Adaptação Dinâmica

Como visto na Seção 2.1.2, na plataforma Meta-ORB a inspeção e a adaptação dinâmica do middleware e das aplicações são realizadas através de reflexão computacional. A plataforma herdou do Open ORB [8] seu *framework* de reflexão multi-modelo. Nessa abordagem, a arquitetura é dividida em um nível base e um meta-nível, que por sua vez é separado em modelos de *meta-espaço* distintos.

Cada modelo de meta-espaço reifica um determinado aspecto do nível base. A arquitetura especifica cinco modelos de meta-espaço, categorizados de acordo com seu estilo de reflexão. Os meta-espaços de *Recursos* e *Interceptação* formam a parte comportamental do meta-nível, enquanto os meta-espaços de *Encapsulamento*, *Interface* e *Arquitetura* formam a parte estrutural. Tipicamente, a reflexão comportamental trata da reificação de aspectos não-funcionais do sistema e a reflexão estrutural trata de aspectos funcionais.

A Figura 2.14 ilustra a composição do meta-nível em diferentes meta-espaços. Cada um dos meta-espaços é implementado como um *meta-componente* distinto responsável por reificar um dos aspectos estruturais ou comportamentais do componente do nível base.

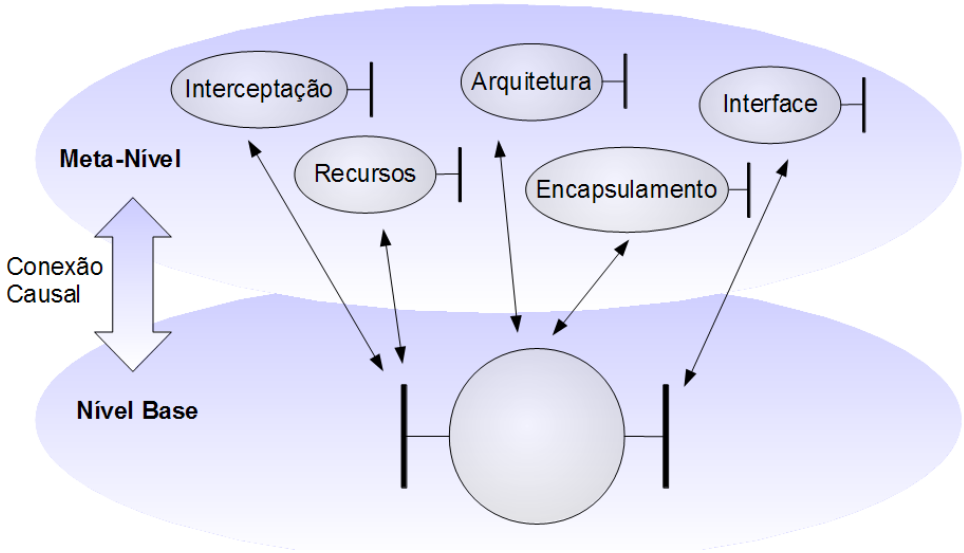



**Figura 2.14:** *Modelo de múltiplos meta-espaços.*

O meta-espaço de *Encapsulamento* oferece uma visão externa de componentes e *bindings*. Esse meta-espaço possibilita apenas a introspecção para descobrir as interfaces expostas por componentes e *bindings* e não altera as entidades do nível base. O meta-espaço de *Interface* provê acesso à estrutura de interfaces individuais. Assim como o meta-espaço de Encapsulamento, esse meta-espaço possibilita apenas a introspecção, com o propósito de descobrir os serviços providos e requeridos e os atributos de uma interface.

O meta-espaço de *Recursos* é responsável por reificar e gerenciar recursos utilizados pelos componentes do nível base. Já o meta-espaço de *Interceptação* possibilita a manipulação de comportamento implícito associado às interfaces, como por exemplo pré- e pós-processamento das interações realizadas por meio de uma interface.

O meta-espaço de *Arquitetura* possibilita, além da introspecção, a adaptação da estrutura interna de componentes e de *bindings* compostos. Esse meta-espaço é a base do mecanismo de reconfiguração dinâmica oferecido pela plataforma Meta-ORB. O meta-

componente Arquitetura não pode ser aplicado a componentes e *bindings* primitivos, pois os mesmos não possuem uma configuração interna e, portanto, não são passíveis de reconfiguração.

Os meta-componentes mantêm uma auto-representação dos objetos do nível base que reificam. A Figura 2.15 ilustra a construção da auto-representação de um *binding*. Primeiramente, o meta-componente obtém do repositório de tipos a definição do *binding*. A definição do *binding* contém a meta-informação que descreve a configuração interna do *binding*, em termos de papéis, interfaces de controle e *bindings* internos. Essa meta-informação é então combinada com informações sobre o *binding* mantidas pelo middleware em tempo de execução, tais como a localização de cada um dos *endpoints* que participam do *binding*. O resultado é compilado em um grafo mantido pelo meta-componente, que contém informações sobre a configuração interna do *binding* em cada um dos *endpoints*.

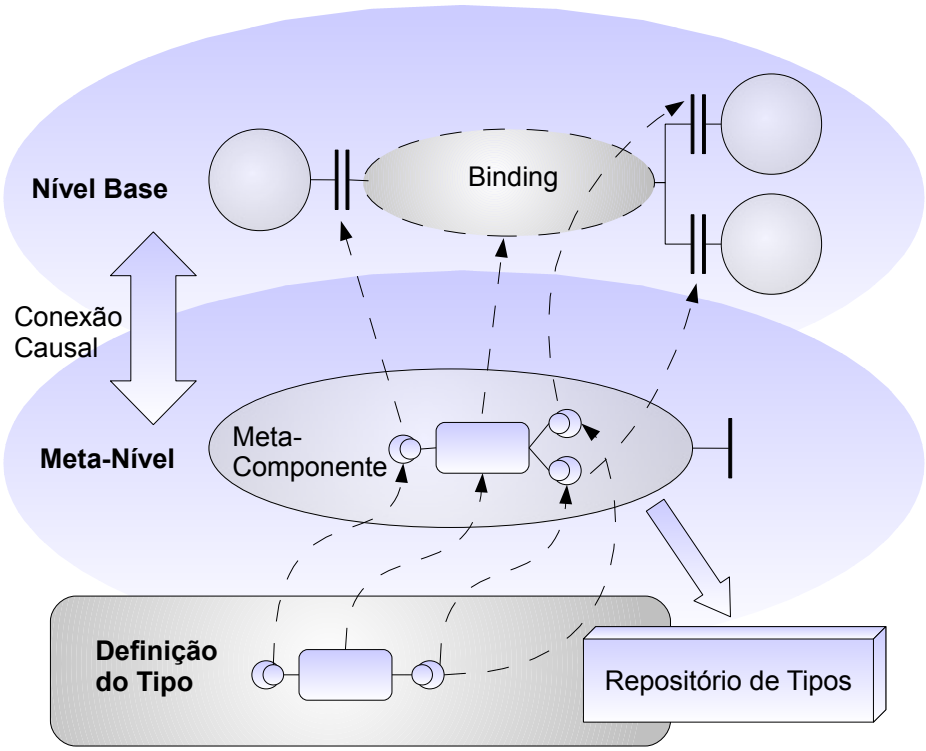



**Figura 2.15:** *Construção da auto-representação de um binding.*

Os meta-componentes não oferecem acesso direto à auto-representação das entidades do nível base para operações de inspeção e adaptação. Essas operações são oferecidas através das interfaces dos meta-componentes, conhecidas como *meta-interfaces*. A meta-interface do meta-componente de Arquitetura, por exemplo, oferece operações que afetam estruturalmente a configuração interna de componentes ou *bindings*, tais como a inserção, remoção e substituição de componentes.

A Figura 2.16 ilustra o meta-componente de Arquitetura *ArchMetaComp* e algumas das operações oferecidas através de sua interface *ArchMetaInterf*. A operação *getObjGraph()* retorna um grafo que pode ser usado para inspecionar a configuração interna de um componente ou *binding*. As operações *insertComponent()* e *removeComponent()* são usadas respectivamente para inserir e remover componentes em um componente composto. As operações *roleInsertComponent()* e *roleReplaceComponent()* são usadas respectivamente para inserir e substituir componentes em todos os *endpoints* que realizam um determinado papel em um *binding*.

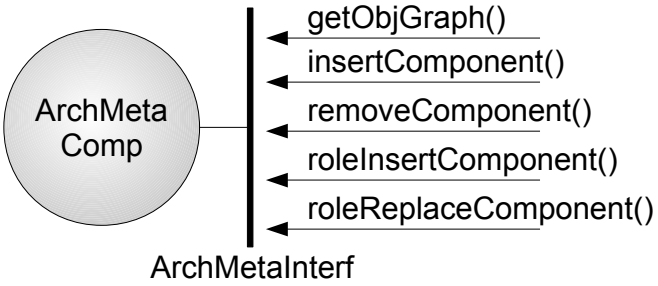



**Figura 2.16:** *Meta-componente de arquitetura.*

Usando o conjunto de operações oferecidas pelas interfaces dos meta-componentes, também conhecido como protocolo de meta-objetos (*Meta-Object Protocol*) [34], o desenvolvedor pode definir programaticamente o comportamento adaptativo das aplicações. O Código 2.1, escrito em linguagem C#, apresenta um exemplo de utilização da interface *ArchMetaInterf* para adaptar o *binding* mostrado anteriormente na Figura 2.13. A estratégia de adaptação implementada define que, sempre que o usuário entra em uma rede domiciliar, não existe mais a necessidade de criptografar as mensagens enviadas neste contexto, e que o componente de criptografia deve ser substituído por um componente que implementa apenas compressão de dados.

**Código 2.1** Exemplo de utilização de uma meta-interface.

```
1.public void onContextChange(object source)
2.{
3.    var archMetaInterf = getArchMetaInterf("chatBinding");
4.    var contextProvider = (ContextProvider)source;
5.
6.    switch(contextProvider.changedContext)
7.    {
8.        case ContextType.Location:
9.            if(contextProvider.newValue == "Home")
10.           {
11.               archMetaInterf.roleReplaceComponent
12.                   ("chatPeerRole", compressorId, "compressor", "crypto");
13.           }
14.           Break;
15.       // tratar outros casos …
16.   }
17.}
```

A operação *roleReplaceComponent* da interface *ArchMetaInterf* é usada no código. Essa operação recebe os seguintes parâmetros:

- O nome do papel afetado (*chatPeerRole*). Apenas os *endpoints* que realizam esse papel são alvo da adaptação;
- O nome que será dado ao novo componente (*compressor*);
- O nome do componente que será substituído (*crypto*);
- O identificador único do tipo do novo componente (*compressorId*), usado para obter a definição desse componente no repositório de tipos.

É importante ressaltar que a maioria desses parâmetros são informações contidas no modelo do *binding*, como mostra a Figura 2.13 e, desta forma, podem ser obtidas pelo programador na definição do tipo do *binding*, contida no repositório de tipos.

## 2.5 Histórico

A arquitetura Meta-ORB foi criada como uma extensão da arquitetura de middleware reflexivo Open ORB [8]. A arquitetura Open ORB é construída sobre duas tecnologias complementares: modelo de programação baseado em componentes e reflexão. Nessa arquitetura, o middleware em si é uma configuração de componentes, que são selecionados durante a compilação do sistema e que podem ser reconfigurados em tempo de execução.

O modelo de componentes de Open ORB é altamente baseado nos pontos de vista de Computação e Engenharia de RM-ODP (*Open Distributed Processing Reference Model*). Nesses pontos de vista, as funções de um sistema são decompostas em objetos, que interagem por meio de interfaces bem definidas. Já seu mecanismo reflexivo é baseado no modelo de meta-espaços particionados [45], em que cada meta-espaço oferece acesso a um aspecto diferente do sistema.

A arquitetura Open ORB foi estendida com um meta-modelo explícito, resultando na arquitetura de middleware reflexivo conhecida como Meta-ORB. Como visto na Seção 2.3, o papel do meta-modelo nessa arquitetura é definir o modelo de programação usado na construção de configurações, tanto da plataforma de middleware quanto das aplicações.

A arquitetura Meta-ORB foi implementada inicialmente em um protótipo escrito em linguagem Python [52], desenvolvido como uma implementação de referência para demonstrar a viabilidade da arquitetura. Mais tarde, um novo protótipo foi desenvolvido, com o objetivo de avaliar a utilização da arquitetura em ambientes móveis e em dispositivos com poucos recursos computacionais, como Palmtops e Pocket PCs. Esse

protótipo foi escrito em linguagem Java, em sua versão Java ME [59], e ficou conhecido como MetaORB4Java [16].

O protótipo de referência escrito em Python e o MetaORB4Java utilizam a mesma implementação do repositório de tipos, também escrita na linguagem Pyhton. Mais tarde, uma nova versão do sistema de tipos foi proposta [18], desta vez baseada na tecnologia EMF (*Eclipse Modeling Framework*) [62].

Para o trabalho apresentado nesta dissertação um novo protótipo da plataforma Meta-ORB foi desenvolvido. O protótipo, chamado de MetaORB.NET, foi escrito em linguagem C# [22], possibilitando o uso das APIs de programação da plataforma .NET [39]. Esse protótipo utiliza a implementação em Java e EMF do repositório de tipos. As implementações do protótipo MetaORB.NET e da nova versão do repositório de tipos serão discutidas com maiores detalhes no Capítulo 4.

## 2.6 Discussão

Este capítulo apresentou a arquitetura da plataforma Meta-ORB, em termos de seu meta-modelo, que define o modelo de programação da plataforma, e de seu mecanismo reflexivo, que possibilita a adaptação dinâmica da plataforma e das aplicações. A plataforma permite o desenvolvimento de aplicações altamente configuráveis e reconfiguráveis. Porém, a adaptação dinâmica deve ser realizada programaticamente, sendo que o desenvolvedor deve definir, no código da própria aplicação, os eventos, condições e ações necessários para a adaptação da mesma, como mostrado anteriormente no Código 2.1. É interessante observar que os eventos que devem ocorrer, as condições necessárias e ações que devem ser tomadas pelo middleware para adaptar a aplicação podem ser vistos como uma política de adaptação. De acordo com políticas definidas localmente, como a do Código 2.1, o desenvolvedor pode instanciar e acessar meta-componentes para adaptar a plataforma e as aplicações.

A implementação local (no código das aplicações) de políticas de adaptação aumenta consideravelmente a complexidade de desenvolvimento de aplicações auto-adaptativas e dificulta o gerenciamento do sistema, principalmente em casos onde existe a necessidade de evoluir as políticas existentes ou adicionar novas políticas. Assim, usando a plataforma Meta-ORB, é possível construir aplicações que se adaptem dinamicamente, embora o procedimento necessário para aplicar as adaptações, bem como as próprias políticas que definem o comportamento adaptativo, se tornam altamente acoplados ao código da aplicação. Esse acoplamento dificulta a manutenção das aplicações, uma vez que é necessário recompilá-las e redistribuí-las sempre que houver mudanças. Conclui-se, desta forma, que a plataforma Meta-ORB precisa de um mecanismo para auto-adaptação que contemple os seguintes requisitos:

- Facilidade na definição das políticas de adaptação independentes do código da aplicação;
- Gerenciamento de políticas de adaptação, possibilitando a adição de novas políticas e a evolução de políticas existentes.
- Mecanismo independente da aplicação para avaliar e executar as políticas de adaptação de forma automática e transparente ao usuário;

Como na plataforma Meta-ORB toda a meta-informação necessária para a criação de uma configuração de middleware é definida com base em seu meta-modelo explícito, é natural que a meta-informação para realização de auto-adaptação também seja definida tendo como base construções presentes no meta-modelo. Isto implica na inserção do conceito de políticas de adaptação no meta-modelo da plataforma Meta-ORB. O Capítulo 3 apresenta uma extensão do meta-modelo para suporte a políticas de adaptação que, juntamente com mecanismos de monitoramento de QoS e gerenciamento de adaptação, tornam a plataforma Meta-ORB auto-adaptativa.

CAPÍTULO 3

# Arquitetura de Auto-Adaptação

Como visto no Capítulo 2, na arquitetura Meta-ORB existe um sistema de tipos que define as entidades (tipos) que podem ser empregadas na construção de configurações customizadas de middleware. Os tipos são descritos de acordo com um meta-modelo explícito e definem a meta-informação utilizada tanto para a configuração estática da plataforma quanto para sua reconfiguração dinâmica. Deste modo, a meta-informação extraída dos tipos também pode ser utilizada por um mecanismo de auto-adaptação do middleware. Para possibilitar a criação de tal mecanismo, o meta-modelo da plataforma deve ser modificado para englobar a meta-informação necessária à auto-adaptação.

Neste capítulo são apresentadas extensões feitas no meta-modelo da plataforma Meta-ORB para inserir o conceito de políticas de adaptação. Com essas extensões, discutidas na Seção 3.1, é possível definir, usando o sistema de tipos da plataforma, as políticas que serão empregadas para adaptar de forma automática o middleware e suas aplicações. Além disso, são apresentadas as modificações feitas na arquitetura da plataforma Meta-ORB para possibilitar a auto-adaptação, que consistem na criação de mecanismos para o monitoramento de QoS, discutido na Seção 3.2, e para o gerenciamento automático de adaptações, discutido na Seção 3.3. Por fim, a Seção 3.4 apresenta algumas considerações sobre a abordagem para auto-adaptação proposta nesse trabalho.

## 3.1 Meta-modelagem de Políticas de Adaptação

Para adicionar o conceito de políticas de adaptação ao meta-modelo da plataforma Meta-ORB, foram criadas novas entidades organizadas no pacote *policies*, mostrado na Figura 3.1. Esse pacote utiliza os elementos básicos do pacote *baseIDL* e é utilizado pelo pacote *componentsAndBindings*. A dependência em relação ao pacote *policies* pelo pacote *componentsAndBindings* indica que os *bindings* podem ser definidos para utilizar certas políticas de adaptação, embora as políticas possam ser definidas de maneira genérica, sem estarem relacionadas a um *binding* específico. Esta abordagem promove a reutilização de políticas em *bindings* distintos.

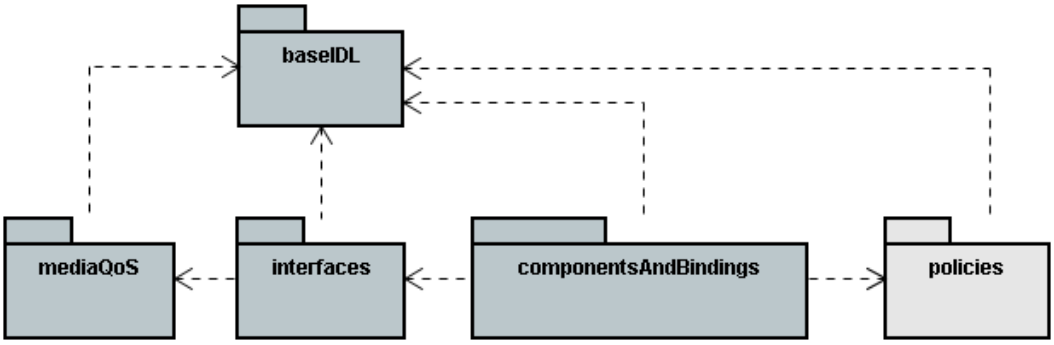



**Figura 3.1:** *Organização do meta-modelo em pacotes: Adição do pacote* ***policies****.*

O pacote *policies* contém dois meta-tipos principais: *AdaptationPolicyDef* e *AdaptationRuleDef*, como mostra de forma simplificada a Figura 3.2. A definição de um *binding* (*BindingDef*) pode estar associada a uma ou mais definições de políticas de adaptação (*AdaptationPolicyDef*). Cada política de adaptação é definida como um *Container* que pode conter um conjunto de definições de regras de adaptação (*AdaptationRuleDef*), que herdam do tipo *Contained* (Conteúdo de um *Container*).

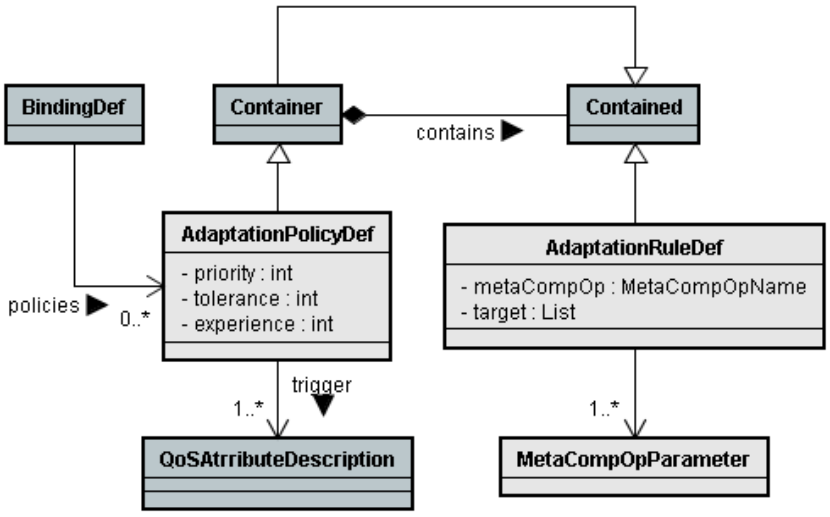



**Figura 3.2:** *Meta-modelagem de políticas e regras de adaptação.*

Neste trabalho, foram consideradas inicialmente apenas adaptações para tratar aspectos de qualidade de serviço presentes nas aplicações distribuídas, principalmente aplicações multimídia e aplicações de tempo real. Deste modo, as políticas de adaptação definem um ou mais atributos de QoS cuja violação dispara a adaptação, conhecidos como gatilhos (*triggers*) da política. Por exemplo, a violação do atributo de QoS que define o atraso máximo tolerável na recepção de um fluxo de mídia pode disparar uma adaptação.

O meta-tipo referenciado como gatilho é *QoSAtrributeDescription*, que apenas caracteriza o atributo de QoS empregado, mas não define os valores de QoS esperados

para esse atributo. Os valores esperados são definidos em atributos de QoS (*QoSAttribute*) associados às operações, sinais ou fluxos das interfaces que contêm anotações de QoS, conforme visto no Capítulo 2, no meta-modelo mostrado na Figura 2.9. A definição de um fluxo de mídia, por exemplo, pode conter anotações de QoS que definem atributos que contêm valores para o atraso máximo tolerável na recepção da mídia.

Os atributos de QoS fazem parte da definição do tipo de uma interface, armazenado no repositório de tipos, e não afetam a interface criada em tempo de execução, sendo utilizados apenas como meta-informação pelo mecanismo de auto-adaptação, que será discutido adiante. Outras interfaces podem conter fluxos cujos valores toleráveis de atraso sejam diferentes, porém a mesma política para tratar violações no atraso pode ser aplicada a todas essas interfaces. Desta forma, as políticas podem ser reutilizadas em *bindings* cujas interfaces participantes definam valores diferentes para os atributos de QoS.

O objetivo do trabalho apresentado nesta dissertação é explorar o auto-gerenciamento de middleware na plataforma Meta-ORB, considerando inicialmente apenas a auto-reconfiguração de componentes. Deste modo, as regras de adaptação expressam apenas ações a serem tomadas para reconfigurar componentes em função de violações nos requisitos de QoS das aplicações. Na plataforma Meta-ORB a reconfiguração dinâmica de componentes é realizada por meio do mecanismo reflexivo da plataforma. Assim, as regras de adaptação definem quais operações do meta-componente associado ao *binding* devem ser empregadas e quais são os valores para os parâmetros dessas operações.

Como no *framework* de reflexão da plataforma Meta-ORB o meta-componente de arquitetura é o único que possibilita, além da introspecção, a adaptação dinâmica da estrutura interna de um *binding*, as operações que podem ser utilizadas nas regras de adaptação limitam-se àquelas disponibilizadas por esse tipo de meta-componente, como as operações mostradas na Figura 2.16 no Capítulo 2. O mecanismo de auto-adaptação pode usar um meta-componente de arquitetura criado para um *binding* que contenha uma ou mais políticas, de forma que as operações de adaptação atuem apenas na configuração interna desse *binding* (componentes e *bindings* internos). Como os *bindings* primitivos não possuem uma configuração interna, não podem existir meta-componentes de arquitetura para os mesmos e, conseqüentemente, também não pode haver políticas de adaptação associadas a esse tipo de *binding*.

Assim como os atributos de QoS, as políticas de adaptação consistem basicamente em meta-informação usada para dirigir o mecanismo de auto-adaptação e não afetam a instanciação das entidades computacionais que formam o *binding* em tempo de execução. Além disso, as políticas são independentes da implementação do mecanismo de auto-adaptação e ficam acessíveis globalmente no repositório de tipos. A utilização do repositório de tipos aumenta o desacoplamento entre políticas e o mecanismo de adaptação. Esse desacoplamento é um princípio chave em um arquitetura adaptativa, uma vez

que, além de possibilitar a reutilização, facilita a identificação de políticas conflitantes e a realização de adaptações coordenadas entre os vários nós do sistema, utilizando políticas globalmente consistentes [23].

A Figura 3.3 ilustra um *binding* explícito que liga um componente produtor (*VideoProducer*) a componentes consumidores (*VideoConsumer*) de vídeo. A definição do tipo desse *binding* (*VideoBinding*), armazenada no repositório de tipos, está associada a uma política de adaptação (*VideoDelayPolicy*), cujo gatilho é um atributo de QoS referente ao atraso na recepção do vídeo (*DelayQoSAttrDesc*). Os componentes consumidores possuem interfaces com fluxos de entrada (*VideoFlowIn*) associados a atributos de QoS (*DelayQoSAttr*) que definem o atraso máximo tolerável na recepção do vídeo. A figura mostra ainda uma das regras de adaptação (*VideoDelayRule*) definida para a política, cujo alvo é o componente *ConsumerStub*. Para completar a política, outra regra semelhante deve ser definida, cujo alvo é o componente *ProducerStub*, embora essa regra não esteja ilustrada na Figura 3.3. Essas regras especificam que a operação *RoleReplaceComponent* do meta-componente deve ser usada para adaptar cada um dos *endpoints* do *binding* que realizam os papéis de consumidores e produtores. Desta forma, o mecanismo de adaptação pode usar a meta-informação contida nos atributos de QoS e nas políticas de adaptação, para verificar violações de QoS e adaptar o *binding* automaticamente.

Como mostra a Figura 3.3, a especificação do comportamento adaptativo passa a fazer parte do modelo da aplicação ou, mais especificamente, da definição dos tipos que formam a configuração do *binding* usado. Assim, as políticas são especificadas declarativamente e não de forma procedural. Desta forma, o projetista da aplicação pode empregar a mesma ferramenta de modelagem para projetar tanto a aplicação quanto seu comportamento adaptativo. Uma ferramenta dessa natureza será apresentada no Capítulo 4.

As políticas de adaptação definem a meta-informação utilizada pelo mecanismo de auto-adaptação da plataforma. Além das regras de adaptação, a definição de uma política contém outros tipos de meta-informação que são utilizados pelo mecanismo de auto-adaptação. A prioridade de uma política (*priority*), o período de tolerância antes de uma adaptação (*tolerance*) e o período de experiência após a adaptação (*experience*), mostrados na Figura 3.3, fazem parte da meta-informação contida em uma política. A finalidade dessa meta-informação será discutida nas próximas seções, que apresentam a arquitetura de auto-adaptação.

## 3.2 Monitoramento de QoS

A arquitetura de auto-adaptação proposta é baseada em políticas de adaptação que agem de acordo com violações em atributos de QoS definidos para as interações

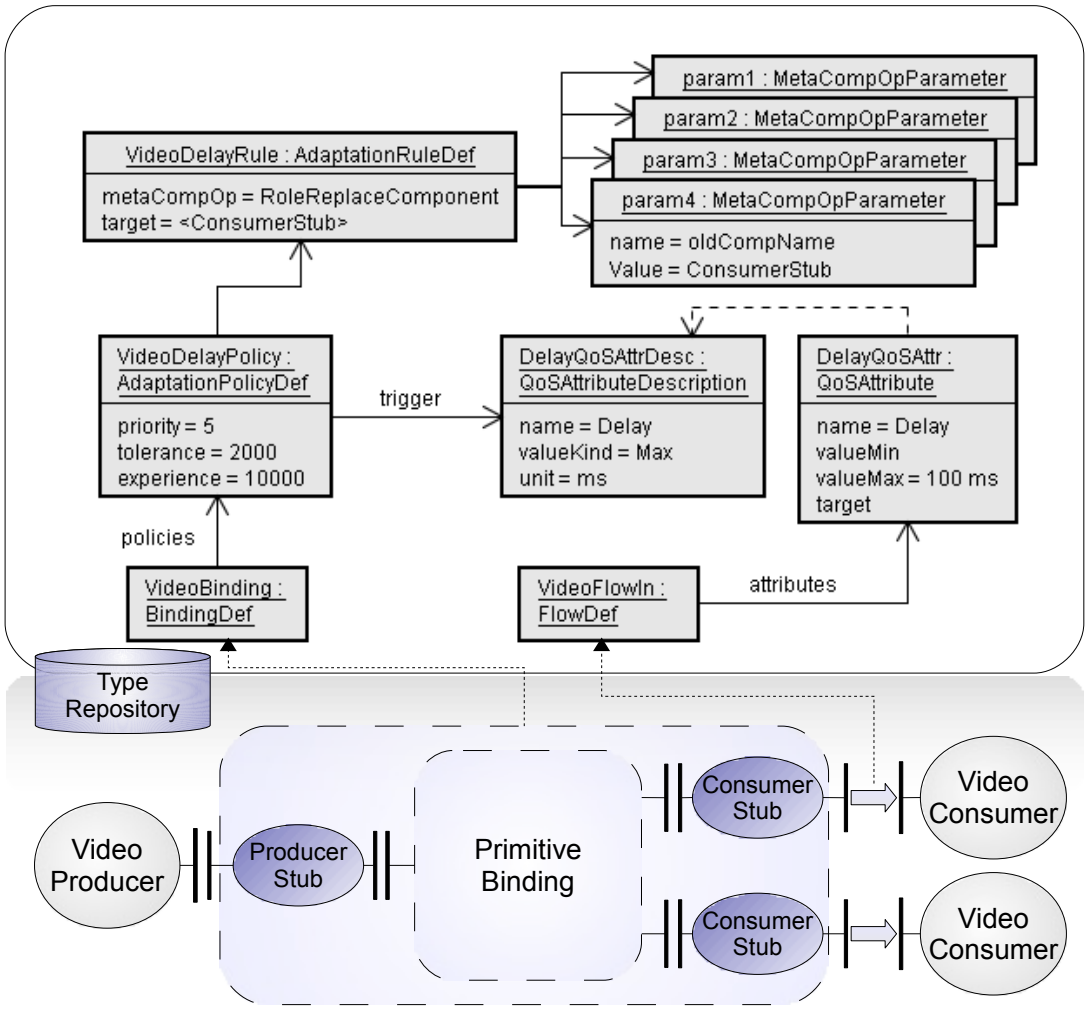



**Figura 3.3:** *Política de adaptação associada a um binding.*

de uma interface. Nesta dissertação, o conceito de qualidade de serviço (QoS) se refere ao conjunto de características quantitativas e qualitativas de um sistema multimídia distribuído, necessárias para seu funcionamento adequado e de acordo com as espectativas dos usuários [64]. Em uma transmissão de vídeo em tempo real, por exemplo, o usuário espera que o atraso não seja muito alto, que áudio e vídeo estejam sincronizados, que não ocorram perdas significativas de trechos de mídia, que a resolução seja suficiente para uma boa visualização, etc.

Cada aplicação possui seu próprio conjunto de requisitos de QoS, que podem ser expressos de forma quantitativa, como, por exemplo, o atraso máximo de 100*ms* (como o esperado pelo consumidor de vídeo da Figura 3.3) ou a resolução mínima de 320$x$200 *pixels*. Em tempo de execução, esses requisitos podem ou não ser atendidos, dependendo das condições do ambiente, como por exemplo a largura de banda disponível para a aplicação. Como não é possível determinar se esses requisitos serão atendidos antes da execução do sistema, as condições do ambiente devem ser constantemente monitoradas.

A manutenção da qualidade de serviço é especialmente importante para aplicações multimídia de tempo-real, em que o tempo é determinante para qualidade da interação entre os usuários. Na arquitetura Meta-ORB, os requisitos de QoS de uma aplicação multimídia distribuída podem ser especificados na definição dos fluxos de mídia presentes na aplicação (*FlowDef* na Figura 2.9). A definição de atributos de QoS para os fluxos oferece um modo de especificar o desempenho esperado quanto às taxas de vazão, atraso, *jitter* e perdas, que são particularmente importantes para a comunicação multimídia em tempo-real [4].

Os atributos de QoS definem as taxas esperadas para determinados parâmetros de QoS, como, por exemplo, o atraso máximo tolerável (*DelayQoSAttr*) mostrado na Figura 3.3. O mecanismo de auto-adaptação deve verificar se os valores dos parâmetros de QoS observados durante a execução da aplicação são aceitáveis de acordo com os valores definidos nos atributos de QoS. Para o *binding* da Figura 3.3, por exemplo, o atraso observado na recepção da mídia em cada um dos consumidores deve ser menor que $100ms$. Para isso, a arquitetura dispõe de um mecanismo baseado no monitoramento do contexto computacional, que permite inferir os valores para os atributos de QoS da aplicação ao longo de sua execução.

Contexto computacional pode ser definido como o conjunto de estados e atributos do ambiente (e.g. qualidade de rede, memória disponível), que determina o comportamento de uma aplicação ou que define um evento de interesse para o usuário da aplicação [12]. Aplicações com requisitos que podem variar de acordo com seu contexto são chamadas de aplicações sensíveis ao contexto. Um exemplo são algumas classes de aplicações para o ambiente móvel, devido ao alto grau de dinamismo deste ambiente, que pode apresentar atributos variáveis de QoS.

Na arquitetura de auto-adaptação proposta nesta dissertação, o mecanismo de monitoramento de QoS é um serviço oferecido pelo middleware, assim como as fábricas de componentes e de *bindings*. Seguindo o modelo de programação da plataforma, o monitor de QoS é um componente abstrato, instanciado pela fábrica de componentes de acordo com uma definição obtida no repositório de tipos. Desta forma, podem existir diferentes implementações para o mecanismo de monitoramento de QoS. Além disso, por ser implementado como um componente, o monitor de QoS pode ser associado a um meta-componente e ter sua estrutura interna reconfigurada dinamicamente.

Uma instância da plataforma pode ser criada de acordo com uma configuração inicial, que especifica os tipos usados em cada serviço do middleware. Essa configuração pode ser descrita em um arquivo contendo os identificadores dos tipos usados em cada um dos serviços. Esse arquivo é lido durante a inicialização do middleware, os tipos são obtidos do repositório por meio de seus identificadores únicos e os componentes que implementam cada um dos serviços são instanciados (Uma implementação concreta

desse processo será discutida no Capítulo 4). O monitor de QoS, assim como os outros componentes que implementam serviços do middleware, é instanciado de acordo com essa configuração inicial.

Cada instância da plataforma possui apenas um componente de monitoramento de QoS. Esse componente pode ser utilizado para o monitoramento de interfaces locais que possuem anotações de QoS. A Figura 3.4 ilustra o componente de monitoramento de QoS agindo sobre interfaces com anotações de QoS. Na figura, as anotações de QoS são ilustradas junto às interfaces em tempo de execução. Porém, essas anotações são obtidas do tipo da interface armazenado no repositório de tipos. A ação do monitor de QoS sobre a interface depende da implementação do componente. Na Seção 3.2.1, será discutida uma abordagem de monitoramento baseada na interceptação das mensagens que passam pela interface.

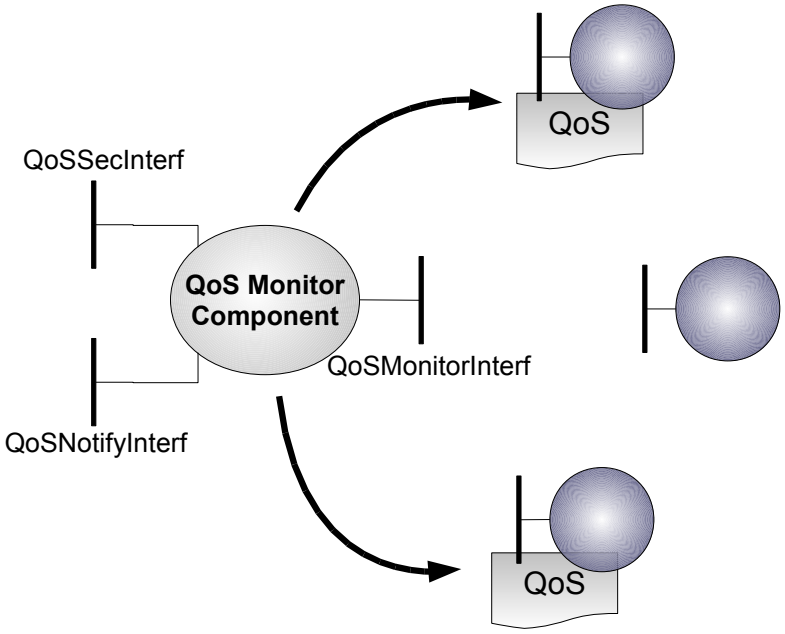



**Figura 3.4:** *Componente de monitoramento de QoS.*

A Figura 3.4 ilustra ainda as interfaces do componente de monitoramento, que são *QoSMonitorInterf*, *QoSSecInterf*, *QoSNotifyInterf*. A interface *QoSMonitorInterf* oferece a outros componentes acesso às funcionalidades do monitor, e as demais interfaces são utilizadas para interação entre monitores de QoS remotos.

Como o monitor de contexto não é um serviço necessário a todas as aplicações desenvolvidas com a plataforma, o componente de monitoramento é instanciado apenas quando requisitado pela primeira vez. Na arquitetura de auto-adaptação, o monitor de QoS é utilizado pelo serviço de gerenciamento de adaptação, que será discutido em detalhes na Seção 3.3. O gerenciador de adaptação está interessado no monitoramento das interfaces com anotações de QoS presentes em um determinado *binding* composto, como o *binding*

da Figura 3.3. Como essas interfaces podem estar localizadas em vários *endpoints* do *binding*, o monitor de QoS deve agir em conjunto com monitores localizados em cada um desses *endpoints*.

A Figura 3.5 mostra a ação do monitor de QoS em conjunto com outros monitores remotos para o monitoramento dos atributos de QoS das interfaces participantes de um *binding*:

1. O monitor de QoS é acessado pela interface *QoSMonitorInterf*, que é uma interface de sinal contendo apenas o sinal *monitor()* que recebe como parâmetro uma referência para a interface a ser monitorada e a definição do atributo de QoS de interesse. Esse monitor passa a ser o monitor primário que compila informações de QoS de todos os *endpoints* do *binding*;
2. O monitor primário verifica se a interface de interesse é uma interface local ou remota. Caso seja uma interface local, o monitor inicia o monitoramento local da interface. Caso seja uma interface remota, o monitor acessa, através de *binding* implícito, a interface *QoSSecInterf* do monitor de QoS localizado no mesmo *endpoint* que a interface remota (monitor secundário), requisitando o monitoramento da mesma. A interface *QoSSecInterf* também é uma interface de sinal, contendo o sinal *monitor()* que recebe como parâmetro o nome único da interface a ser monitorada, a definição do atributo de QoS de interesse, e o nome único do monitor primário que solicitou o monitoramento;
3. Cada um dos monitores secundários busca localmente a interface especificada como parâmetro e inicia seu monitoramento de acordo com o atributo de QoS também passado como parâmetro;
4. Caso algum dos monitores secundários identifique a violação de algum dos atributos de QoS especificados, ele notifica o monitor primário através da interface *QoSNotifyInterf*, também acessada por *binding* implícito. O *binding* implícito é criado a partir da resolução no serviço de nomes do nome único do monitor primário passado como parâmetro ao monitor secundário. Através do sinal *notifyViolation()* da interface *QoSNotifyInterf*, o monitor primário recebe informações sobre os valores medidos que geraram uma violação de QoS nas interfaces monitoradas. O monitor primário pode então compilar as informações de QoS de todas as interfaces de interesse presentes no *binding* e informar o gerenciador de adaptação.

### 3.2.1 Monitoramento Baseado em Interceptadores

A implementação concreta do componente de monitoramento de QoS deve conter as estratégias para coleta das informações de contexto referentes aos atributos

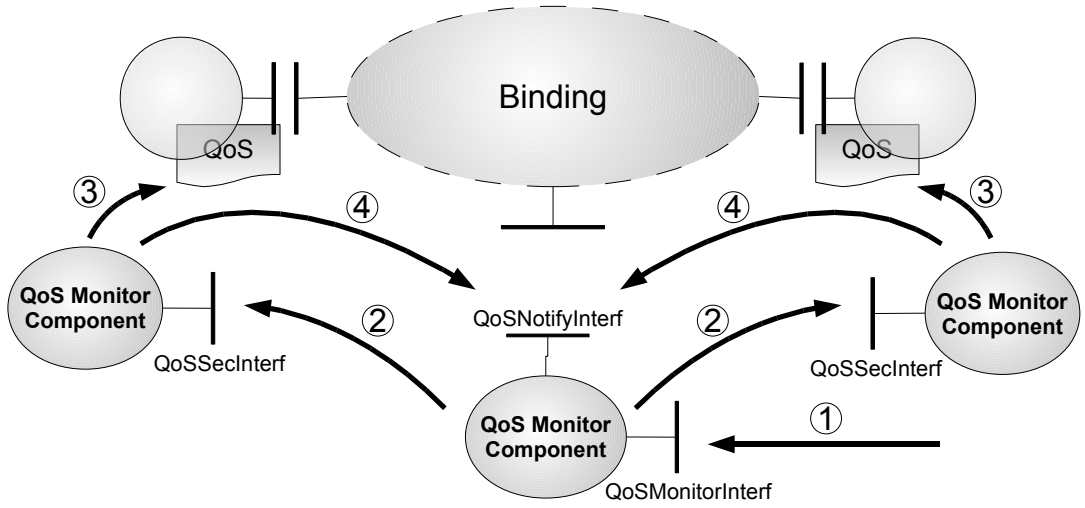



**Figura 3.5:** *Monitoramento de QoS nas interfaces participantes de um binding.*

de QoS passados como parâmetro. Podem existir diversas informações de contexto relevantes a uma aplicação [12]:

- Contexto computacional - informações de conectividade da rede e dos recursos computacionais disponíveis, assim como outras informações sobre hardware e software obtidas por meio de sensores;
- Contexto do usuário - informações sobre o usuário referentes a dados pessoais e algumas informações sensoriadas, como sua localização;
- Contexto físico - informações sobre o ambiente exterior, como luminosidade, temperatura, etc;
- Contexto temporal - como a data e hora atual.

Como a proposta de uma máquina de provisão de informações de contexto está além do escopo deste trabalho, o mecanismo de monitoramento de QoS foi definido de maneira abstrata. Assim, implementações concretas do componente para monitoramento de QoS podem utilizar ou estender abordagens existentes, como [21] e [56]. Porém, para efeito de avaliação da arquitetura, algumas estratégias para o monitoramento das informações de contexto computacional, necessárias ao estudo de caso que será apresentado no Capítulo 5, foram implementadas e serão discutidas a seguir.

As estratégias de monitoramento empregadas utilizam o mecanismo reflexivo da plataforma, através do meta-espaço de interceptação, para inserir interceptadores que coletam informações de QoS nas interfaces. Uma implementação do meta-espaço de interceptação é apresentada no Capítulo 4, Seção 4.2.3. A Figura 3.6 ilustra a utilização do meta-componente de interceptação para o monitoramento dos atributos de QoS:

1. Uma requisição para o monitoramento de uma interface é feita ao monitor de QoS através da interface *QoSMonitorInterf* (monitor primário) ou da interface *QoSSecInterf* (monitor secundário);
2. O atributo de QoS passado como parâmetro na requisição é passado a um seletor de estratégias de monitoramento interno do monitor. Cada estratégia é implementada como um interceptador criado para monitorar um atributo de QoS específico. O desenvolvedor de estratégias de monitoramento deve implementar o interceptador seguindo uma padronização de nomes. Por exemplo, para o atributo *DelayQoSAttr* deve existir um interceptador chamado *DelayQoSAttrInterceptor*;
3. Com o nome do atributo de QoS, o seletor pode derivar o nome do interceptador e verificar se ele existe. Caso o seletor encontre uma estratégia para o atributo (um interceptador adequado), a estratégia é instanciada. A instanciação de uma estratégia consiste em acessar o meta-componente de interceptação da interface e requisitar a inserção de um interceptador para coletar dados a respeito do atributo;
4. O meta-componente insere o interceptador na interface;
5. Os interceptadores monitoram o fluxo de chamadas que passam pela interface e passam ao monitor de contexto informações a respeito do atributo monitorado.

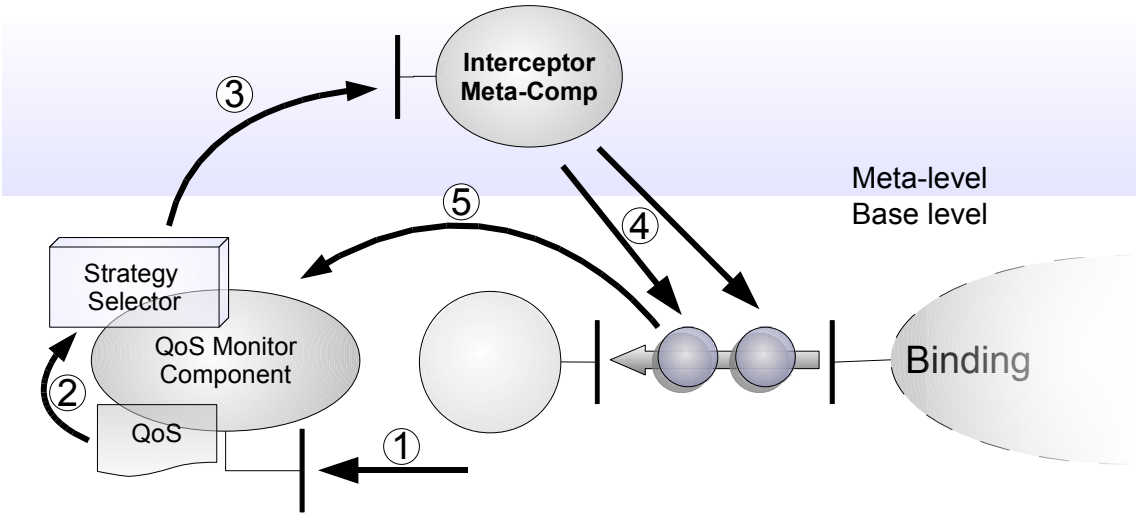



**Figura 3.6:** *Interceptadores para monitoramento de QoS.*

O monitoramento de QoS baseado na interceptação do fluxo que passa por uma interface pode ser empregado para coletar um conjunto pequeno de informações de contexto a respeito da interação feita com o uso do *binding*. Para outros tipos de informação de contexto que podem ser úteis para adaptação, como a localização do usuário e o nível da bateria do dispositivo, outras abordagens de monitoramento precisam ser implementadas. Porém, algumas das principais informações de QoS relevantes a

aplicações multimídia de tempo real, como o nível de atraso e de perdas de pacotes, podem ser obtidas por meio de interceptadores. Portanto, para avaliar a arquitetura proposta em uma implementação concreta, essa abordagem é suficiente, uma vez que, para o estudo de caso do Capítulo 5, as informações de contexto sobre o atraso e as perdas de pacotes são suficientes.

## 3.3 Gerenciamento de Adaptações

Assim como o monitoramento de QoS, o gerenciamento de adaptações é um serviço oferecido pelo middleware por meio de um componente abstrato. Do mesmo modo, o gerenciador de adaptações é instanciado de acordo com a configuração inicial do middleware, que especifica o tipo do componente concreto que será empregado. A função desse componente é interpretar as meta-informações contidas nas políticas de adaptação associadas a um *binding* e executar um protocolo de auto-adaptação.

O componente de gerenciamento de adaptações é instanciado sob demanda, caso seja necessário o monitoramento de QoS e a auto-adaptação de um *binding*. A Figura 3.7 ilustra a inicialização do componente de gerenciamento de adaptações:

1. Durante a criação de um *binding*, a fábrica de *bindings* utiliza a meta-informação contida na definição do *binding* (obtida no repositório de tipos) para instanciar cada um dos *endpoints* do *binding*;
2. Quando finaliza a criação do *binding*, a fábrica verifica se a definição do *binding* possui alguma definição de política de adaptação. Caso existam políticas de adaptação para o *binding* em questão, a fábrica acessa o componente de gerenciamento de adaptação pela interface operacional *AMInterf*. Essa interface contém a operação *register()*, que recebe como parâmetro o nome único do *binding* e uma lista com as políticas de adaptação associadas a ele. Como o *binding* pode ser formado por outros *bindings* internos, o processo é repetido para cada um dos *bindings* internos que contenham políticas de adaptação;
3. A partir do nome único do *binding*, o gerenciador de adaptação inicializa um meta-componente de arquitetura para o *binding*. Com esse meta-componente, conectado causalmente ao *binding*, o gerenciador pode tanto inspecionar quanto adaptar dinamicamente a estrutura interna do *binding*.

Após o registro do *binding* e suas políticas no gerenciador de adaptação, um protocolo de auto-adaptação é iniciado. Inicialmente, o gerenciador verifica se existem políticas de adaptação conflitantes. Existem dois casos que caracterizam as políticas como conflitantes: quando possuem o mesmo atributo de QoS como gatilho e quando a adaptação tem o mesmo alvo.

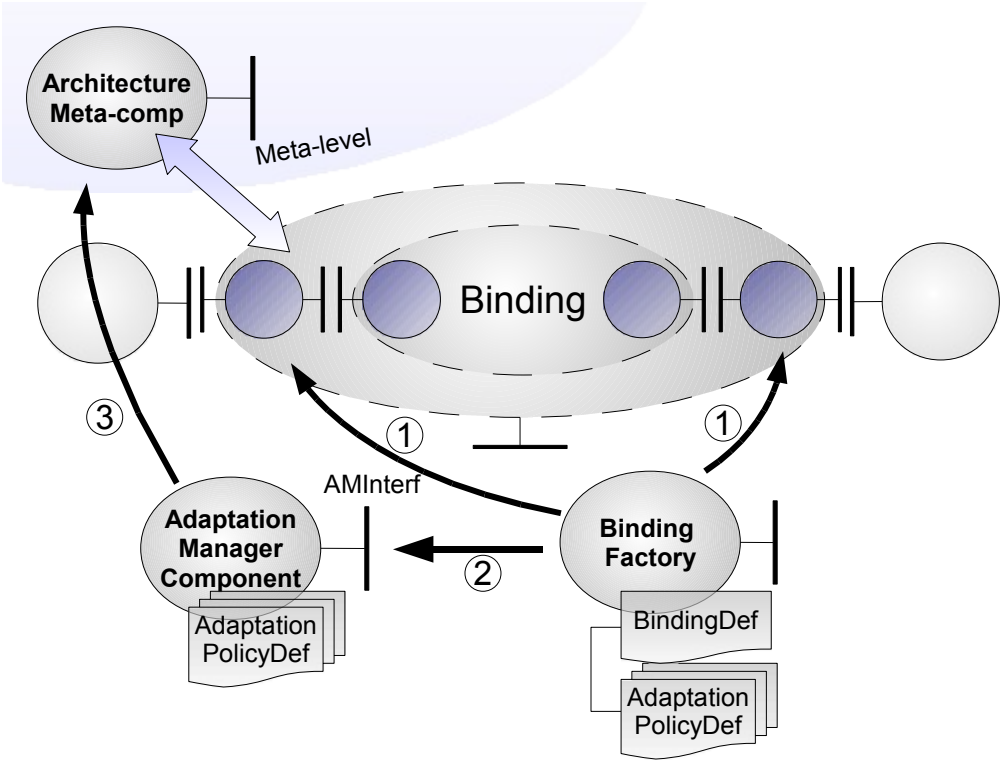



**Figura 3.7:** *Inicialização do gerenciador de adaptação.*

No primeiro caso, o gerenciador deve verificar os atributos de QoS que disparam cada uma das políticas (*trigger*) para identificar aquelas que tratam as mesmas violações de QoS. Por exemplo, pode existir mais de uma política para tratar violações referentes ao atraso no *binding* da Figura 3.3, ou seja, políticas distintas contendo *DelayQoSAttrDesc* como gatilho. Neste caso, mesmo que não sejam conflitantes de acordo com o segundo caso, as políticas não podem ser aplicadas em conjunto pois não há garantias de que o efeito esperado por uma não seja influenciado por outra.

No segundo caso, o gerenciador deve verificar o alvo das regras de adaptação, para identificar políticas que realizam adaptações estruturais em um mesmo ponto de uma configuração. Os alvos de uma adaptação (*target* no meta-modelo da Figura 3.2) são especificados na definição do tipo de uma regra de adaptação. No caso da inserção de um componente, o alvo é o *binding* local onde o componente será inserido. No caso de remoção ou substituição de componentes, os alvos são o próprio componente e os *bindings* locais que ligam suas interfaces, que serão removidos ou substituídos. Por exemplo, podem existir duas políticas para o *binding* da Figura 3.3 cujo alvo das regras é o componente *ConsumerStub*. Neste caso, as duas políticas não podem ser aplicadas em conjunto, pois o componente *ConsumerStub* pode ter sido substituído ou removido pela primeira política aplicada, de modo que a segunda política não possui mais um alvo.

As políticas podem possuir vários gatilhos, assim como várias regras de adap-

tação que afetam o diversos pontos em uma configuração. Para simplificar o protocolo, não foram consideradas as situações envolvendo tais políticas, de modo que, mesmo que uma política defina mais de um gatilho, se apenas um deles estiver em conflito com o gatilho de outra política, as duas são consideradas conflitantes. Deste modo, o projetista do *binding* deve tentar associá-lo apenas a políticas que não sejam conflitantes, ou definir prioridades para as mesmas, como será discutido a seguir.

Cada política possui uma prioridade, especificada na definição de seu tipo, como pode ser visto na Figura 3.3. Caso existam conflitos, o gerenciador deve utilizar as políticas com maior prioridade, sendo que as demais são armazenadas em uma lista de políticas reserva, que podem ser utilizadas no futuro. Caso existam políticas conflitantes com a mesma prioridade, o gerenciador pode escolher qualquer uma delas, e a outra vai para a lista de espera. A implementação de um algoritmo de checagem de conflitos é apresentada no Capítulo 4.

Uma vez checados os conflitos e descartadas as políticas conflitantes, o gerenciador inicia o monitoramento dos atributos de QoS. Utilizando o meta-componente, o gerenciador de adaptação obtém uma representação da configuração atual do *binding*, contendo informações sobre cada um dos *endpoints* instanciados, possibilitando a inspeção dos mesmos. O gerenciador de adaptação identifica então quais interfaces participam do *binding*, ou seja, busca no repositório de tipos a definição dos tipos das interfaces dos componentes externos ligados ao *binding* e dos componentes e papeis internos ao *binding*. A partir das definições dos tipos, é possível determinar quais interfaces possuem anotações de QoS contendo os mesmos atributos especificados como gatilhos das políticas.

Conhecendo os atributos de QoS cuja violação dispara cada uma das políticas de adaptação e a localização das interfaces com anotações de QoS, o gerenciador inicia o monitoramento de QoS para o *binding*. O gerenciador acessa o monitor de QoS local através da interface *QoSMonitorInterf*, requisitando o monitoramento das interfaces do *binding*, de acordo com o processo mostrado na Figura 3.5. O monitor coleta dados em todos os *endpoints* do *binding* que contenham interfaces com anotações de QoS e informa ao gerenciador de adaptação caso ocorra alguma violação.

A Figura 3.8 mostra a ação do gerenciador de adaptação em decorrência da violação dos atributos de QoS monitorados:

1. O monitor de QoS notifica uma violação de QoS em uma das interfaces monitoradas ao gerenciador de adaptação, usando a interface *AMNotifyInterf*;
2. O gerenciador avalia a política associada ao atributo de QoS violado. Como a violação pode ser em decorrência de uma condição temporária do ambiente, cada política define um período de tolerância (*tolerance* no meta-modelo da Figura 3.2) durante o qual o gerenciador espera para verificar se a violação é temporária ou

se ela se mantém. Na política mostrada na Figura 3.3, por exemplo, o período de tolerância é de 2000*ms*;

3. Caso a violação continue, o gerenciador salva os últimos dados coletados pelo monitor (para posterior comparação) e aplica as regras de adaptação. Cada regra possui as informações necessárias para a invocação de uma operação do meta-componente associado ao *binding*. A política mostrada na Figura 3.3, por exemplo, possui uma regra que define que a operação *RoleReplaceComponent* do meta-componente deve ser utilizada, bem como os parâmetros que devem ser passados para essa operação.
4. O meta-componente realiza então a adaptação da configuração do *binding*, o que pode consistir na inserção de novos componentes ou, como ilustra a Figura 3.8, na substituição de componentes existentes.

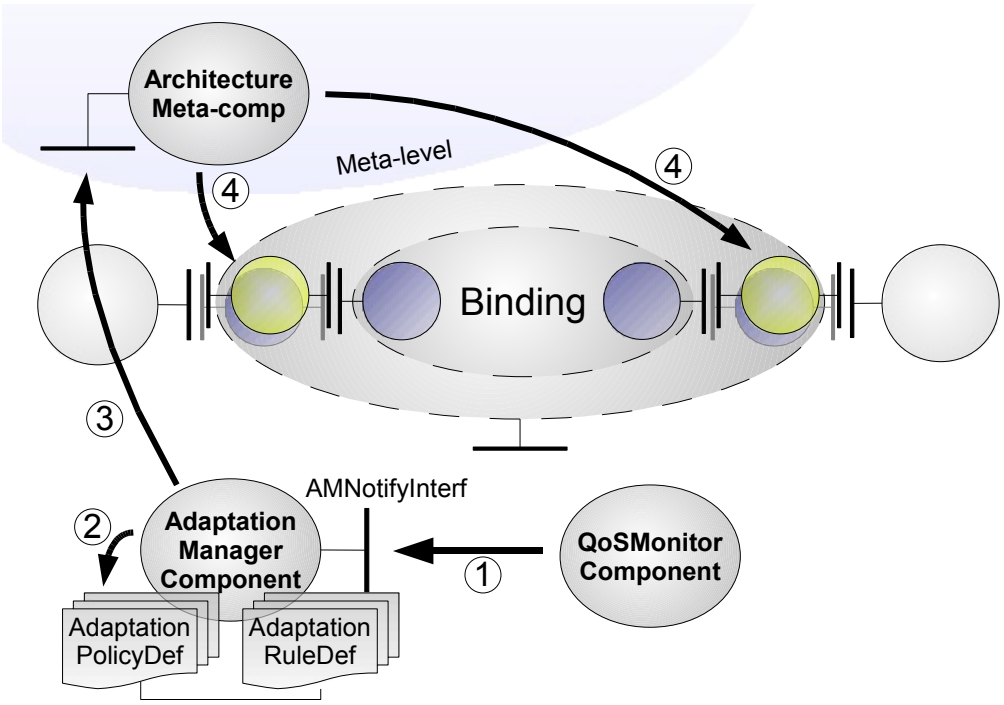



**Figura 3.8:** *Gerenciamento de adaptação.*

O gerenciador de adaptação não checa a consistência das regras de adaptação, ou seja, não são checados se os parâmetros passados às operações do meta-componente são válidos. O próprio meta-componente realiza essa checagem. Então, caso um parâmetro seja inválido, como, por exemplo, o nome incorreto do componente a ser substituído ou o identificador incorreto do tipo de um componente a ser inserido, o meta-componente dispara uma exceção e a adaptação é cancelada. Esse tipo de inconsistência nas políticas ou regras de adaptação pode ser evitado caso a checagem seja feita na própria ferramenta de modelagem (definição) de tipos, que será discutida no Capítulo 4.

Após a adaptação do *binding*, ele entra em período de experiência, durante o qual o gerenciador avaliará se a adaptação obteve um resultado satisfatório, de acordo com os valores de QoS monitorados. A duração do período de experiência (*experience* no meta-modelo da Figura 3.2), assim como do período de tolerância, é obtida na definição do tipo da política de adaptação. Para a política mostrada na Figura 3.3, por exemplo, o período de experiência é de 10000*ms*. O gerenciador não pode aplicar outras políticas em um *binding* que está em período de experiência. Durante esse período, caso não seja mais notificado sobre violações dos atributos de QoS tratados pela política, o gerenciador considera que a adaptação foi bem sucedida e continua sua operação para as políticas que restaram.

Durante o período de experiência, caso continue sendo notificado sobre a violação dos atributos de QoS monitorados, o gerenciador compara os dados de monitoramento atuais com os dados armazenados antes da adaptação. Se, apesar da violação ter ocorrido novamente, os dados atuais forem melhores que os anteriores à adaptação, o gerenciador considera a adaptação válida. Por exemplo, considerando um atributo de QoS que define um atraso máximo de 100*ms*. Antes da adaptação, o valor monitorado foi de 200*ms*. Após a adaptação, esse valor diminuiu para 150*ms* (o que ainda constitui uma violação, embora com um valor melhor que os 200*ms* anteriores) sendo a adaptação considerada válida. Isso ocorre para garantir um ganho, mesmo que pequeno, em relação ao atributo de QoS.

Se os valores monitorados forem piores que os anteriores à adaptação, o gerenciador considera a adaptação inválida e solicita ao meta-componente que desfaça a reconfiguração. Por exemplo, considerando o cenário anterior, onde o atraso máximo é de 100*ms* e o valor monitorado antes da adaptação é de 200*ms*. Se após a adaptação, esse valor aumentar para 250*ms*, ou se mantiver em 200*ms*, significa que, além de não fazer efeito, a adaptação prejudicou ainda mais a aplicação e deve ser desfeita. A política inválida é desconsiderada pelo gerenciador, que pode escolher uma nova política para ser aplicada dentre as política de prioridade mais baixa que estão na lista de espera, caso exista alguma.

O gerenciamento de adaptação e o monitoramento dos atributos de QoS continua durante toda a execução da aplicação ou até o *binding* ser desfeito. O protocolo de auto-adaptação não considera cenários onde, após a adaptação, os valores de QoS observados no ambiente retornam a um estado onde uma adaptação realizada anteriormente não seja mais necessária. Assim, uma adaptação válida é mantida até o fim da execução da aplicação.

O gerenciador possibilita ainda a inclusão dinâmica de políticas de adaptação para *bindings* que já estejam sendo gerenciados. Isso pode ser feito com a operação *register()* da interface *AMInterf*, passando como parâmetro o nome único do *binding* e as novas políticas a serem incluídas. Neste caso, o gerenciador repete o processo de interpretação

para as novas políticas incluídas. Atualmente, esse processo é possível apenas programaticamente. Porém, trabalhos futuros podem considerar a inclusão automática de políticas no gerenciador em função de mudanças realizadas nos tipos armazenados no repositório de tipos.

## 3.4 Considerações

Este capítulo apresentou uma arquitetura para auto-adaptação baseada na plataforma de middleware reflexivo Meta-ORB. A arquitetura utiliza políticas de adaptação como meta-informação para dirigir o mecanismo de auto-adaptação. Para isso, o conceito de políticas de adaptação foi inserido no meta-modelo da plataforma. Assim, as políticas passam a fazer parte do modelo da aplicação, descrito em termos de tipos armazenados no repositório de tipos. Desta forma, as políticas podem ser definidas de acordo com o meta-modelo e com o apoio da mesma ferramenta utilizada na definição dos demais elementos que formam a configuração de uma aplicação.

Na abordagem proposta, o desenvolvedor de uma aplicação auto-adaptativa não precisa dominar diferentes linguagens e tecnologias, podendo criar um modelo da aplicação, contendo seus requisitos de QoS e políticas de adaptação, através de uma linguagem de modelagem unificada e usando uma única ferramenta. Esse modelo é usado diretamente pelo middleware como meta-informação para instanciar e gerenciar a aplicação, sem a necessidade de sua tradução para uma linguagem de programação específica. Além disso, como os modelos podem ser gerenciados por um repositório global, é possível reutilizar definições de componentes, *bindings* e políticas de adaptação em novas aplicações ou mesmo evoluir e controlar versões de modelos existentes.

Na proposta original da plataforma Meta-ORB, o comportamento adaptativo é descrito no código das aplicações de maneira procedural, como por exemplo no Código 2.1. Na arquitetura proposta, esse comportamento é descrito de maneira declarativa, utilizando políticas de adaptação, e realizado por um mecanismo autônomo independente da aplicação. As políticas declarativas possuem uma sintaxe simplificada fixa, seguindo o meta-modelo da plataforma, o que facilita sua especificação pelo desenvolvedor. Já o mecanismo de auto-adaptação, implementado nos componentes de monitoramento de QoS e gerenciamento de adaptação, foi inserido na infra-estrutura básica do middleware e não precisa ser carregado ou habilitado pelo usuário ou desenvolvedor.

O Capítulo 4 apresenta uma implementação concreta da arquitetura de middleware auto-adaptativo proposta, desenvolvida para demonstrar a viabilidade da abordagem proposta. A implementação consiste em um protótipo funcional da plataforma Meta-ORB, chamado MetaORB.NET, que engloba, além das características descritas no Capítulo 2, o mecanismo de auto-adaptação descrito neste capítulo.

CAPÍTULO 4

# Implementação

Como discutido brevemente no Capítulo 2, um novo protótipo da plataforma Meta-ORB foi desenvolvido para validar a arquitetura de auto-adaptação proposta nesta dissertação. Esse protótipo, chamado MetaORB.NET, foi escrito na linguagem de programação C# e utiliza a plataforma .NET. Este capítulo discute os principais aspectos envolvidos na implementação desse protótipo.

A implementação consistiu no desenvolvimento da infra-estrutura básica de middleware mais as extensões para auto-adaptação, resultando em uma versão auto-adaptativa da plataforma Meta-ORB. Essa implementação foi utilizada no desenvolvimento da aplicação apresentada como estudo de caso, que será discutida no Capítulo 5. O restante deste capítulo é estruturado como segue. A Seção 4.1 apresenta a implementação do sistema de tipos, que é a base para o gerenciamento de meta-informações na plataforma. A Seção 4.2 apresenta o protótipo MetaORB.NET, abordando características relevantes da implementação que permitiram a construção da arquitetura auto-adaptativa. A Seção 4.3 discute alguns dos aspectos importantes da implementação da infra-estrutura de auto-adaptação. Por fim, a Seção 4.4 apresenta considerações sobre a implementação, bem como melhorias que podem ser realizadas em trabalhos futuros.

## 4.1 Sistema de Tipos

O sistema de tipos representa o meta-modelo da plataforma Meta-ORB, definindo as construções que podem ser empregadas na modelagem de uma configuração do middleware. Com um meta-modelo bem definido (nível 2), é possível criar ferramentas para a definição, armazenamento e recuperação dos tipos (modelo, nível 1). A ferramenta utilizada para esse propósito é o repositório de tipos. Os tipos, por sua vez, são utilizados pelas fábricas especializadas (fábricas de componentes e *bindings*) para a instanciação de configurações do middleware (entidades de tempo de execução, nível 0).

A primeira atividade de implementação realizada no contexto dessa dissertação foi o desenvolvimento de um repositório de tipos baseado na tecnologia EMF (*Eclipse Modeling Framework*) [62]. EMF consiste em um *framework* de modelagem e geração

de código que pode ser empregado na construção de ferramentas e outros tipos de aplicações baseadas em um modelo bem estruturado. Uma discussão sobre o emprego dessa tecnologia para a criação do repositório de tipos pode ser encontrada em [18].

EMF pode ser visto como uma implementação em Java de um subconjunto do meta-modelo de MOF (*Meta Object Facility*)[42], embora especifique seu próprio meta-modelo, chamado *Ecore*. Com EMF, e seus *plugins* para a plataforma Eclipse [62], é possível especificar um modelo e gerar as classes de implementação desse modelo em linguagem Java. Os modelos podem ser especificados em XMI (*XML Metadata Interchange*) [44], que é o formato padrão para a definição de modelos em EMF. Modelos podem, também, ser importados a partir de modelos descritos em UML (criados com alguma ferramenta de modelagem), a partir de interfaces Java anotadas ou a partir de esquemas XML.

Seguindo o paradigma de MDE (*Model Driven Engineering*) [57], com EMF o modelo de uma aplicação deixa de ser usado apenas para documentação e passa a ser usado diretamente por ferramentas de produção do software [6]. Como visto no Capítulo 2, a distinção entre modelo e meta-modelo é apenas uma questão de ponto de vista, sendo que um meta-modelo pode ser visto como o modelo que descreve um outro modelo. Assim, é possível utilizar EMF para gerar automaticamente ferramentas de produção de modelos de middleware a partir de um meta-modelo bem estruturado. Neste caso, o que é utilizado pelo gerador de códigos de EMF é um modelo *Ecore*, que descreve o meta-modelo do middleware.

A Figura 4.1 mostra um modelo *Ecore*, criado com o *plugin* de EMF para a plataforma Eclipse, que descreve o meta-modelo da plataforma Meta-ORB. A figura mostra cada um dos pacotes do meta-modelo da plataforma (introduzidos anteriormente na Figura 3.1), bem como as entidades contidas no pacote *policies* (Figura 3.2). A partir desse modelo, o código do sistema de tipos da plataforma pode ser gerado automaticamente, contendo classes Java que representam cada uma das construções do meta-modelo. A instanciação dessas classes produz as entidades do modelo que podem ser usadas para construir uma configuração particular de middleware, ou seja, os tipos.

Além do código de implementação de um modelo, EMF possibilita a geração de ferramentas gráficas para a criação e edição de instâncias desse modelo. Essas ferramentas são *plugins* que podem ser integrados à plataforma Eclipse e utilizados nesse ambiente de desenvolvimento. A Figura 4.2 mostra um modelo (definição de tipos) criado com a ferramenta gráfica gerada a partir do meta-modelo da plataforma Meta-ORB. Nessa figura, o modelo é estruturado em diversos módulos, cada um contendo a definição dos tipos que representam uma parte da configuração do middleware e das suas aplicações. O módulo *ORBcore*, por exemplo, contém os tipos que definem os serviços básicos do núcleo da plataforma. Já o módulo *AdaptationPolicies*, contém a definição das políticas de

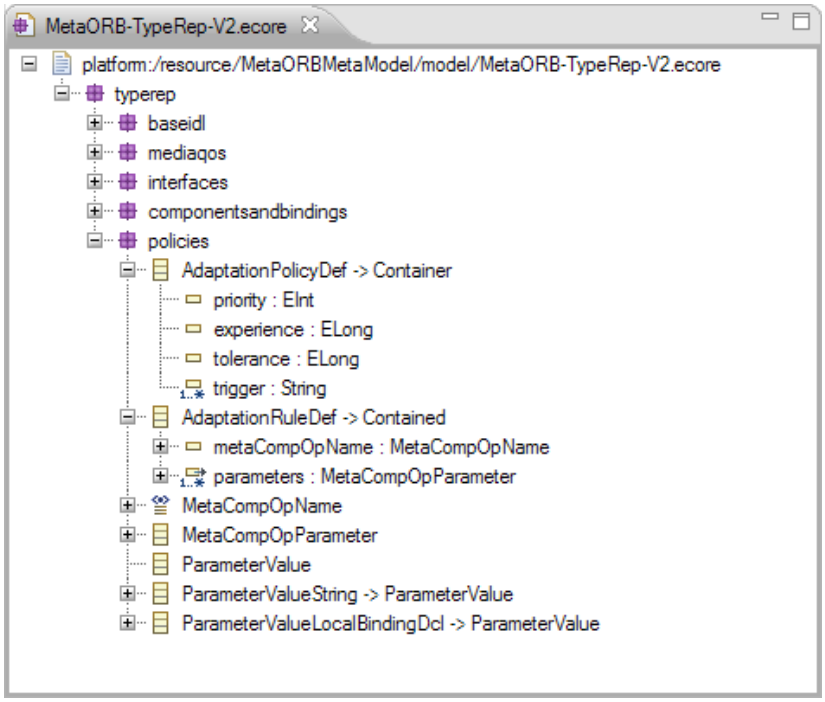



**Figura 4.1:** *Meta-modelo da plataforma Meta-ORB em Ecore.*

adaptação, e o módulo *SimpleBindingDefs* contém a definição de alguns tipos de *bindings*.

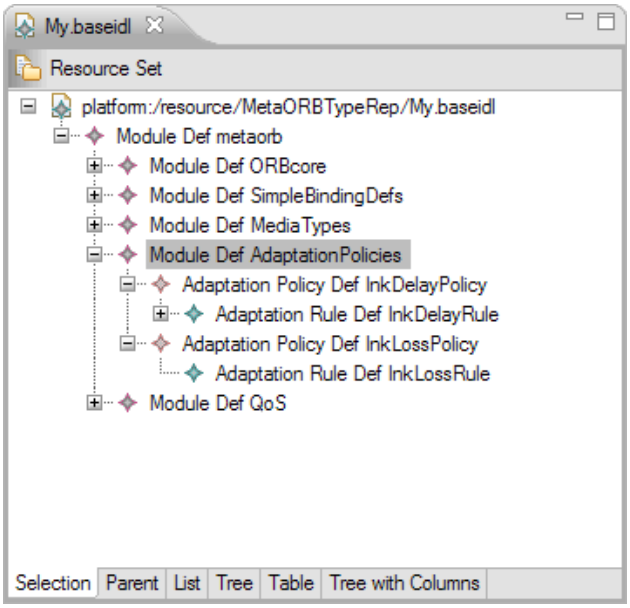



**Figura 4.2:** *Ferramenta de edição de modelos.*

O repositório de tipos propriamente dito foi implementado como uma API de acesso aos tipos criados com a ferramenta gráfica e armazenados em um arquivo. O repositório oferece operações como *lookup()*, que recebe como parâmetro o identificador único de um tipo e retorna sua definição. A definição do tipo é retornada em formato XMI, que é independente de linguagem de programação, possibilitando a interpretação

do tipo por versões da plataforma que não sejam implementadas em Java, como o próprio MetaORB.NET.

Um serviço web foi desenvolvido para fornecer acesso ao repositório de tipos para instâncias remotas da plataforma. Esse serviço é descrito em WSDL (*Web Services Description Language*) [14]. Essa linguagem permite a descrição estruturada, independente de linguagem de programação, da interface do serviço, dos tipos de dados utilizados e de sua localização. Com essa descrição, é possível criar um serviço em Java que pode ser acessado por clientes implementados tanto em Java quanto em C#.

A Figura 4.3 ilustra a utilização do repositório de tipos através do serviço web:

1. O código do sistema de tipos e o *plugin* gráfico para edição de tipos são gerados a partir do meta-modelo da plataforma. O *plugin* pode então ser empregado na definição dos tipos que formam uma configuração de middleware. Essa definição é salva em um arquivo no servidor que irá executar o serviço remoto do repositório de tipos;
2. Um cliente remoto, escrito em Java ou em C#, acessa o serviço web do repositório e faz uma consulta por um tipo com método *lookup()*. A chamada é repassada ao processo que executa o repositório de tipos localmente;
3. O repositório faz uma busca pelo tipo no arquivo que contém as definições de tipos. Em tempo de execução, o tipo encontrado é uma instância da classe *EObject*, do modelo Ecore de EMF;
4. O repositório serializa o tipo em formato XMI e o retorna para o serviço web, que por sua vez, o retorna para o cliente remoto.

O Código 4.1 mostra a definição de um componente serializada em XMI. Essa definição contém a meta-informação necessária para a fábrica de componentes instanciar o componente e suas interfaces. O componente em questão é o componente primitivo *BFComp*, que corresponde à fábrica de *bindings*. A definição contém, entre outras informações, o nome da classe de implementação do componente (*implem_name*) e o identificador único de suas interfaces (*interfType*). Deste modo, a fábrica pode instanciar a classe de implementação do componente e buscar a definição de cada uma das interfaces para criá-las.

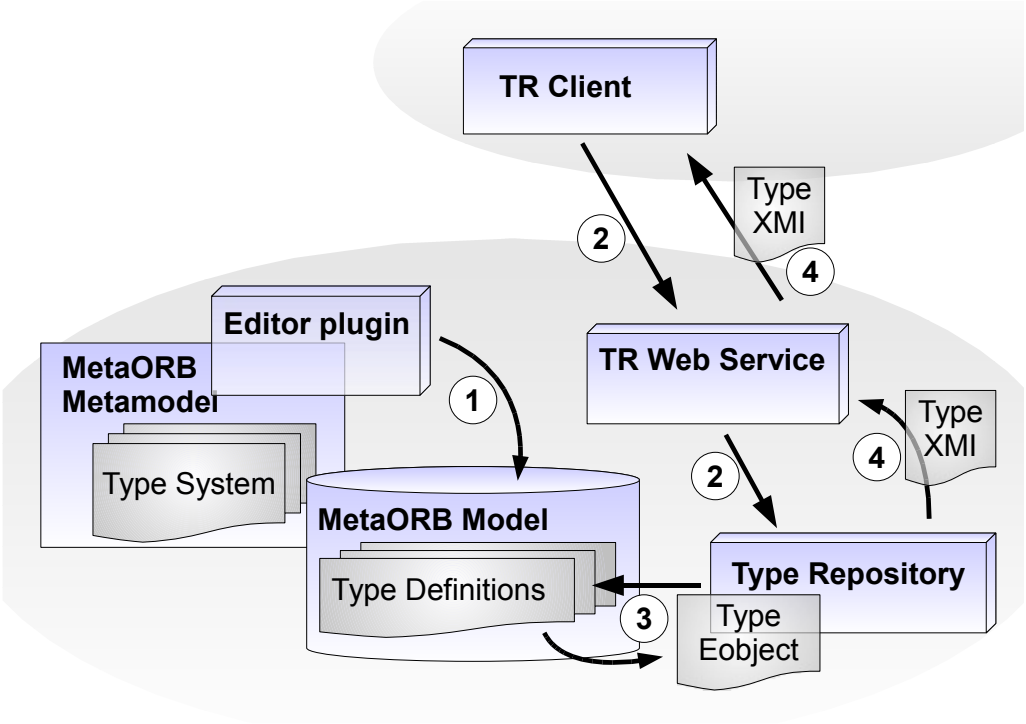



**Figura 4.3:** *Repositório de Tipos: Servidor.*

**Código 4.1** Definição do tipo de um componente em XMI.

```
<?xml version="1.0" encoding="us-ascii"?>
<typerep.componentsandbindings:PrimComponentDef
    name="BFComp"
    id="idl:inf.ufg.br/metaorb/ORBcore/SimpleBF/BFComp:1.0"
    absolute_name="metaorb/ORBcore/SimpleBF/BFComp"
    implem_name="BindingFactory">
    <interfaces interf_name="BF">
        <interfType
            href="idl:inf.ufg.br/metaorb/ORBcore/SimpleBF/BFInterf:1.0"/>
    </interfaces>
    <interfaces interf_name="BFSec">
        <interfType
            href="idl:inf.ufg.br/metaorb/ORBcore/SimpleBF/BFSecInterf:1.0"/>
    </interfaces>
    <interfaces interf_name="BFCollect">
        <interfType
            href="idl:inf.ufg.br/metaorb/ORBcore/SimpleBF/BFCollectInterf:1.0"/>
    </interfaces>
</typerep.componentsandbindings:PrimComponentDef>
```

O Código 4.2 mostra (de maneira simplificada) a definição de uma das interfaces do componente *BFComp*, a interface operacional *BFInterf*. Segundo a definição, essa interface realiza o papel de servidor (*ROLE_SERVER*) e provê apenas uma operação, a operação *newB*. A operação *newB*, usada para criar um *binding* explícito, recebe como parâmetros uma lista de referências para as interfaces que serão ligadas pelo *binding* (*irefList*), o identificador único do tipo (*BindingDefId*) e o nome dado ao *binding*

(*BindingName*). O tipo de cada um dos parâmetros foi omitido do código. Além disso, a operação *newB* pode lançar uma exceção cujo tipo é especificado na *tag exceptionDef*.

**Código 4.2** Definição do tipo de uma interface em XMI.

```
<?xml version="1.0" encoding="us-ascii"?>
<typerep.interfaces:OpInterfaceDef
    name="BFInterf"
    id="idl:inf.ufg.br/metaorb/ORBcore/SimpleBF/BFInterf:1.0"
    absolute_name="metaorb/ORBcore/SimpleBF/BFInterf"
    is_abstract="false"
    role="ROLE_SERVER">
    <contents
        xsi:type="typerep.interfaces:OperationDef"
        name="newB"
        id="idl:inf.ufg.br/metaorb/ORBcore/SimpleBF/BFInterf/newB:1.0"
        absolute_name="metaorb/ORBcore/SimpleBF/BFInterf/newB">
        <exceptionDef
           href="../../MetaORB/TypeRep/My.baseidl#idl:
                 inf.ufg.br/metaorb/ORBcore/ORBcoreException:1.0"/>
        <params name="irefList"/>
        <params name="BindingDefId"/>
        <params name="BindingName"/>
    </contents>
</typerep.interfaces:OpInterfaceDef>
```

A Figura 4.4 ilustra o acesso ao repositório de tipos, feito por uma fábrica durante a criação de um componente:

1. A fábrica de componentes busca a definição do tipo de um componente usando um serviço local do middleware, que é a interface de acesso ao repositório de tipos global. O serviço local verifica se o tipo não está armazenado em uma *cache* contendo os tipos buscados anteriormente. Caso não esteja, o serviço local repassa a requisição para o cliente do serviço remoto, que faz a consulta no repositório de tipos global;
2. O tipo é retornado em formato XMI para o cliente, que o repassa ao serviço local. O serviço local converte o tipo, atualmente em XMI (formato primitivo *string*), para um objeto da linguagem de implementação da plataforma (Java ou C#). No protótipo em Java, o tipo pode ser convertido para um objeto do tipo *EObject*, de acordo com as classes geradas pelo EMF a partir do meta-modelo. Já no protótipo em C# o tipo é convertido em um objeto do tipo *XElement*, classe que representa um elemento XML, facilitando a navegação por seus nós e atributos;
3. O tipo é repassado à fábrica de componentes, que irá interpretá-lo. O tipo provê a meta-informação necessária para a instanciação do componente;
4. O componente e suas interfaces são construídos em tempo de execução de acordo com a definição do tipo obtida do repositório.

Essa implementação do repositório de tipos foi integrada inicialmente apenas ao protótipo MetaORB.NET. A integração com a plataforma MetaORB4Java ainda precisa

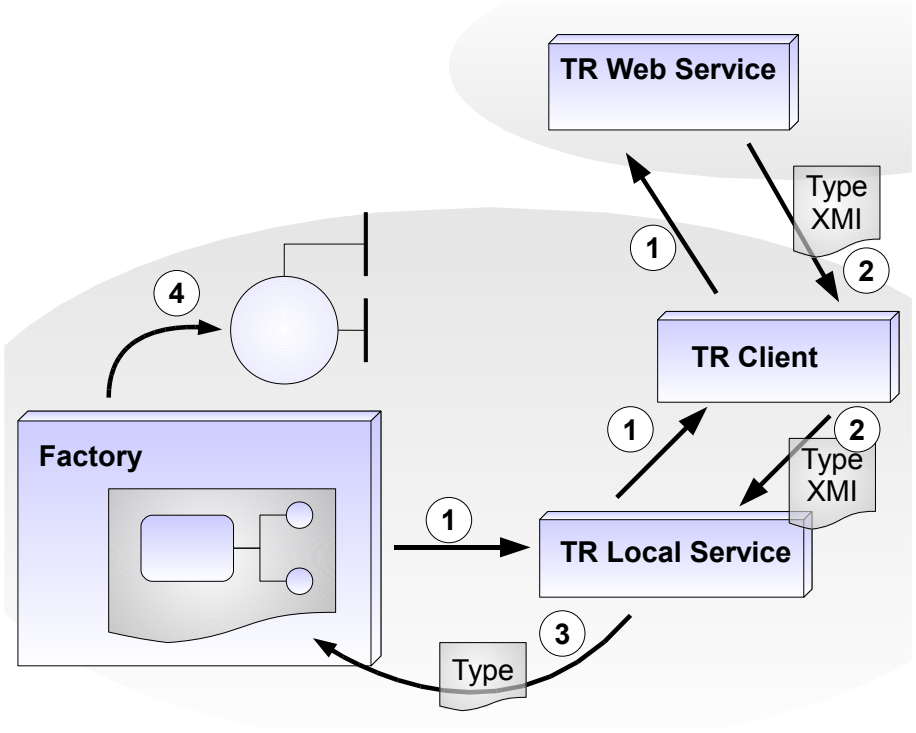



**Figura 4.4:** *Repositório de Tipos: Cliente.*

ser realizada, assim como um estudo da interoperabilidade entre os dois protótipos. Dentre os benefícios da implementação atual, podem ser citados:

- uso de ferramentas baseadas em EMF, que facilitam a manipulação do meta-modelo e a definição dos tipos;
- acesso através de um serviço web, que possibilita a utilização do repositório por protótipos distintos, implementados em diferentes linguagens de programação; e
- a utilização do formato XMI, que torna os tipos independentes de linguagem de programação.

## 4.2 Protótipo MetaORB.NET

O protótipo MetaORB.NET é uma implementação em linguagem C#, para a plataforma .NET, da arquitetura Meta-ORB e suas extensões para auto-adaptação. Grande parte da implementação das funcionalidades do middleware, como os protocolos para a criação de componentes e de *bindings*, foi baseada nos protótipos anteriores, escritos em Python e Java. Uma discussão sobre a implementação de cada um desses protótipos pode ser encontrada em [15] e [16, 48] respectivamente. Deste modo, nesta seção serão abordados apenas aspectos de implementação que não estão presentes nesses outros trabalhos que descrevem os protótipos anteriores da plataforma Meta-ORB.

Em sua implementação atual, a plataforma MetaORB.NET oferece as seguintes funcionalidades:

- *Núcleo* - Suporte ao modelo de programação da plataforma Meta-ORB, com construções que representam, em tempo de execução, as entidades de primeira classe, como componentes, interfaces e *bindings*. Além disso, contém a infra-estrutura básica do middleware e a implementação padrão de componentes especializados, como as fábricas de componentes e de *bindings*;
- *Serviços Remotos* - Serviços globais, como o repositório de tipos e o serviço de nomes, que, para manter a interoperabilidade com a plataforma MetaORB4Java, foram implementados em Java e acessados através de serviços web;
- *Meta-nível* - Implementação dos mecanismos reflexivos da plataforma. Neste protótipo, esses mecanismos são realizados por componentes que implementam os modelos de meta-espaço de arquitetura, encapsulamento e interceptação;
- *Infra-estrutura de auto-adaptação* - Implementação da arquitetura apresentada no Capítulo 3, composta pelo gerenciador de adaptação e pelo monitor de QoS, utilizando o modelo de programação e as demais funcionalidades da plataforma.

### 4.2.1 Cápsula

Assim como nos protótipos anteriores, os serviços do middleware são expostos por um objeto especial do núcleo da plataforma, chamado de *Cápsula*. Uma cápsula, instanciada em um dispositivo, representa a unidade de localização para todos os objetos (componentes, interfaces e *bindings*) criados em seu contexto dentro de um ambiente distribuído. As cápsulas, durante sua inicialização, utilizam um arquivo de configuração inicial para criar os serviços básicos do middleware. Esse arquivo contém a localização do serviço de nomes e do repositório de tipos, assim como os identificadores únicos dos tipos que definem os componentes que implementam os demais serviços do middleware.

A Figura 4.5 ilustra uma cápsula e alguns dos serviços básicos oferecidos pelo middleware. Inicialmente, a cápsula inicializa os clientes do repositório de tipos e do serviço de nomes, que não são serviços realizados como componentes locais. Possuindo acesso ao repositório de tipos, a cápsula busca a definição dos componentes que implementam os demais serviços do middleware. O primeiro componente criado, por um procedimento fixo, é a fábrica de componentes. Para a criação dos demais componentes, a cápsula passa a empregar a fábrica de componentes.

Dentre os componentes mostrados na Figura 4.5 estão os componentes da arquitetura de auto-adaptação, que são o gerenciador de adaptação (*AdaptationManager*) e o monitor de QoS (*QoSMonitor*). Esses componentes, como visto no Capítulo 3, são criados sob demanda e, portanto, não são instanciados na inicialização da cápsula. Outro

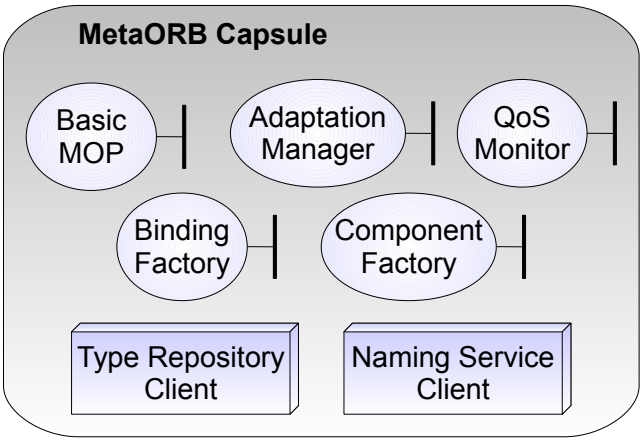



**Figura 4.5:** *Cápsula no protótipo MetaORB.NET.*

componente mostrado na figura é o componente *BasicMOP*, usado para obter acesso aos meta-componentes que reificam objetos locais ou remotos. O componente *BasicMOP* oferece operações que recebem como parâmetro o nome único de um objeto (componente, interface ou *binding*) e retorna uma referência para a interface (local ou remota) do meta-componente associado a esse objeto. A sigla MOP se refere ao protocolo de meta-objetos (*Meta-Object Protocol*) da plataforma.

### 4.2.2 Modelo de Programação

Para implementar o protótipo MetaORB.NET, o modelo de programação da plataforma Meta-ORB foi reimplementado na linguagem C#. Essa linguagem foi usada para aumentar domínio das aplicações que podem ser desenvolvidas com a plataforma, que passam a contar com recursos específicos dessa linguagem, especialmente as APIs oferecidas pela plataforma .NET, como a própria API de tinta digital usada no estudo de caso do Capítulo 5. Nessa implementação, assim como nas anteriores, as construções de primeira classe do modelo de programação passam a ser representadas por objetos da linguagem. Deste modo, as fábricas de componentes e de *bindings* devem interpretar os tipos dos componentes, interfaces e *bindings* e instanciar as classes C# que representam componentes, interfaces e *bindings* em tempo de execução.

#### Componentes e Interfaces

As principais construções do modelo de programação são os componentes e suas interfaces. A Figura 4.6 mostra um diagrama contendo as classes que implementam essas construções na linguagem C#. De acordo com o diagrama, os componentes podem ser primitivos (*PrimComponent*) ou compostos (*Component*). Neste protótipo, apenas o

protocolo de criação de componentes primitivos foi implementado, visto que esse tipo de componente é o suficiente para validar a arquitetura de auto-adaptação.

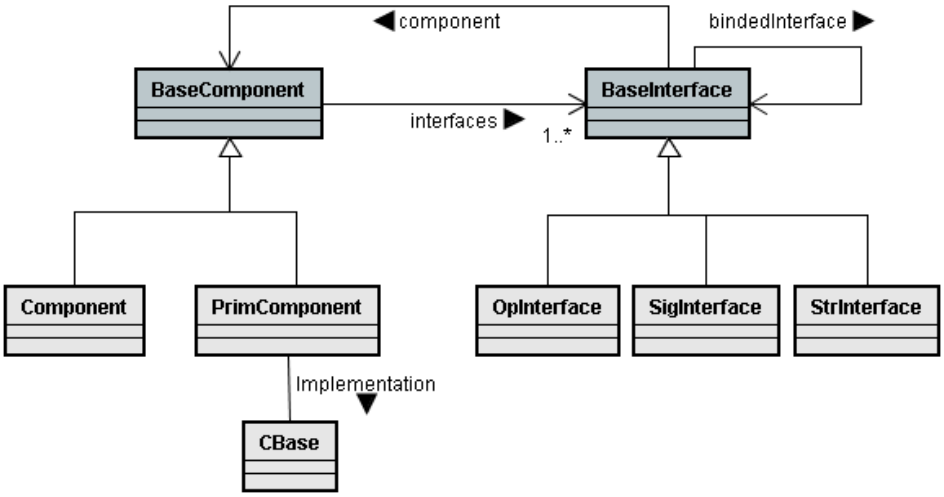



**Figura 4.6:** *Classes de implementação para componentes e interfaces.*

A fábrica de componentes, de posse da definição de um componente primitivo, como aquela mostrada no Código 4.1, cria o componente em tempo de execução instanciando inicialmente suas interfaces. Diferente das interfaces de objetos da linguagem (interfaces C#), que definem apenas a assinatura dos métodos implementados pelo objeto, as interfaces Meta-ORB definem, além da assinatura dos métodos, como se dará a interação do componente através da interface. Deste modo, uma interface Meta-ORB é implementada por meio de uma interface C# contendo as assinaturas dos métodos e uma classe contendo a implementação de seu comportamento, sendo instanciada como um objeto em tempo de execução.

O Código 4.3, por exemplo, mostra a definição da interface *AMInterf* do componente gerenciador de adaptação, obtida no repositório de tipos e serializada em XMI. Segundo essa definição, a interface *AMInterf* é uma interface operacional (*OpInterfaceDef*), que realiza o papel de um servidor (*ROLE_SERVER*), ou seja, define operações providas pelo componente. A interface define apenas a operação *register*, que é uma operação de sentido único (*OP_ONEWAY*), ou seja, não possui um valor de retorno. A operação *register* define os parâmetros *bindingName* e *policiesIds*, referentes ao nome do *binding* que será registrado no gerenciador de adaptação e o identificador único dos tipos das políticas de adaptação que serão empregadas. Além disso, uma definição de interface pode conter outras meta-informações (que não são mostradas no Código 4.3), como os tipos de retorno das operações e os tipos de cada um dos parâmetros.

**Código 4.3** Definição do tipo de uma interface em XMI.

```
<?xml version="1.0" encoding="ASCII"?>
<typerep.interfaces:OpInterfaceDef
    name="AMInterf"
    id="idl:inf.ufg.br/metaorb/ORBcore/SimpleAM/AMComp/AMInterf:1.0"
    absolute_name="metaorb/ORBcore/SimpleAM/AMComp/AMInterf"
    is_abstract="false"
    role="ROLE_SERVER">
    <contents xsi:type="typerep.interfaces:OperationDef"
        name="register"
        id="idl:inf.ufg.br/metaorb/ORBcore/SimpleAM/AMComp/AMInterf/register:1.0"
        absolute_name="metaorb/ORBcore/SimpleAM/AMComp/AMInterf/register"
        mode="OP_ONEWAY">
        <params name="bindingName"/>
        <params name="policiesIds"/>
    </contents>
</typerep.interfaces:OpInterfaceDef>
```

A definição de interface do Código 4.3 corresponde à classe mostrada no Código 4.4, que representa a interface *AMInterf* na linguagem C#. Essa classe herda da classe *OpInterface*, que é a super-classe para todas as interfaces operacionais. A implementação da operação *register* consiste em chamar o método *invoke* da super-classe. Esse método determina o comportamento da operação, segundo as anotações que definem o papel da interface (*Role*) e seu modo de operação (*OperationMode*). Como o papel da interface é o de um servidor (*InterfRole.SERVER*), a operação será invocada na implementação do componente. Como o modo de operação é de sentido único (*OperationMode.ONEWAY*), nenhum resultado é retornado.

**Código 4.4** Implementação em C# de uma interface de componente.

```
[Role(InterfRole.SERVER)]
public class AMInterf : OpInterface, IAMInterf
{
    public AMInterf(string InterfName, object Component, string Id)
        : base(InterfName, Component, Id)
    {
    }

    public AMInterf(BaseInterface Ibase)
        : base(Ibase)
    {
    }

    #region IAMInterf Members

    [OperationMode(OperationMode.ONEWAY)]
    public void register(string bindingName, string[] policiesIds)
    {
        var args = new object[] { bindingName, policiesIds };
        invoke(GetType().GetMethod("register"), args);
    }

    #endregion
}
```

O método *invoke*, implementado pela classe *OpInterface*, simplesmente decide se a operação será invocada no componente ou em uma interface ligada por *binding* lo-

cal. Nos outros estilos de interface, mostrados na Figura 4.6, de sinal (*SigInterface*) e de fluxo contínuo (*StrInterface*), o comportamento também é determinado por anotações que definem a direção dos sinais ou fluxos, mas a implementação do método *invoke* é diferente. Para a implementação das interfaces de sinal, são utilizadas chamadas assíncronas, cujo retorno é imediato e não existe bloqueio para espera do processamento do método chamado.

Já as interfaces de fluxo contínuo possibilitam a interação por fluxos (*stream*) de mídia. As interfaces de *stream* que definem fluxos de saída podem ser ativadas e passar a enviar automaticamente amostras de mídia produzidas pelo componente para um *binding* local. Nessas interfaces, o método que representa um fluxo de saída precisa ser chamado apenas uma vez, tendo uma coleção de amostras de mídia como parâmetro. Sempre que a coleção recebe uma nova amostra, a interface a envia automaticamente. Assim como os sinais, os fluxos são assíncronos.

Uma vez instanciadas as interfaces, a classe de implementação do componente primitivo é instanciada. Essa classe deve herdar da classe *CBase* e implementar as interfaces C# que definem a assinatura dos métodos de cada uma das interfaces do componente. Cada um dos métodos que representam operações providas, como o *register* da interface *AMInterf*, bem como aqueles que representam sinais de entrada ou fluxos de entrada, devem conter a implementação de uma funcionalidade do componente. Já as operações requeridas, sinais de saída ou fluxos de saída são implementados apenas como um redirecionamento para as respectivas interfaces.

### Bindings

A plataforma Meta-ORB define três tipos de *binding*: local, implícito e explícito. Os *bindings locais* são realizados no contexto de uma única cápsula e consistem apenas em uma troca de referências entre os objetos em tempo de execução, as quais representam as interfaces ligadas. Como mostrado na Figura 4.6, uma instância da classe *BaseInterface* contém uma referência para outra instância de *BaseInterface*, representando um *binding* local.

Um *binding* local pode ser criado com o uso de um método especial da Cápsula (*localBinding()*), que recebe como parâmetros duas interfaces locais, ou por meio de um componente especializado. Esse componente faz parte dos serviços básicos do middleware e é criado durante a instanciação da cápsula, podendo ser utilizado na realização de *bindings* locais entre duas interfaces locais ou entre duas interfaces remotas (localizadas no contexto de uma única cápsula). O protótipo MetaORB.NET possibilita apenas o *binding* local entre interfaces estritamente compatíveis. Deste modo, em um *binding* local entre interfaces operacionais por exemplo, para cada operação provida por uma das interfaces deve existir uma operação requerida correspondente na outra interface.

Os *bindings implícitos* são criados através da resolução dos nomes únicos armazenados no serviço de nomes, o que resulta em uma referência para uma interface remota. O Código 4.5 mostra a utilização do serviço de nomes, através do método *lookup()*, para obter uma referência para a interface registrada com o nome *AMInterf:AMComp:Capsule1* , que é a interface do tipo *AMInterf* de um gerenciador de adaptação (*AMComp*) localizado em uma cápsula remota (*Capsule1*). As chamadas são enviadas à interface remota através de um mecanismo de comunicação básico do middleware que, assim como outros serviços do middleware, é implementado como componentes especializados criados durante a inicialização da cápsula. Atualmente, esses componentes implementam um mecanismo de comunicação do tipo cliente/servidor usando o protocolo TCP e a serialização binária padrão da plataforma .NET para enviar as mensagens.

**Código 4.5** Resolução do nome único de uma interface resultando em um binding implícito.

```
1.IAMInterf amIRef =
2.    capsule.NameService.lookup<AMInterf>("AMInterf:AMComp:Capsule1");
3.amIRef.register(bindingUname, policiesIDs);
```

Nos protótipos anteriores da plataforma, o método *lookup()* retornava um objeto do tipo *IRef*, que consiste em uma estrutura que provê informações sobre a interface e sua localização. Para obter uma referência ativa, que permite o acesso à interface remota através de *binding* implícito, o método *resolve()* do serviço de nomes era usado. No protótipo atual, existe o método *lookup()*, que retorna uma referência ativa para uma interface remota, e o método *lookupType()*, que retorna uma lista de referências (*IRefs*) para todas as interfaces de um certo tipo registradas no serviço de nomes. O método *lookup()* é usado principalmente para obter acesso às interfaces remotas por meio de *binding* implícito. Já o método *lookupType()* foi criado principalmente para facilitar a obtenção da lista de referências utilizada na criação de *bindings* explícitos.

Os *bindings explícitos* consistem em *bindings* formados por configurações internas de componentes e outros *bindings*, e são criados explicitamente para ligar duas ou mais interfaces remotas. A criação de *bindings* explícitos é realizada por um componente especializado da cápsula, a fábrica de *bindings*. A definição de um tipo de *binding* descreve sua configuração interna, formada por componentes e outros *bindings* que realizam um determinado papel de acordo com a interface que será ligada ao *binding*. A função da fábrica de *bindings* é interpretar o tipo do *binding* e instanciar a configuração de cada um dos papéis em cada um dos *endpoints* que participam do *binding*. A implementação da fábrica de *bindings* para o protótipo MetaORB.NET utiliza o mesmo protocolo de criação de *bindings* utilizado nos protótipos anteriores [15, 48] e por isso não será detalhada nesta dissertação.

### 4.2.3 Meta-Nível

Como discutido anteriormente, no protótipo MetaORB.NET a implementação do meta-nível reflexivo conta com os meta-componentes de encapsulamento, arquitetura e interceptação. Esses três meta-componentes são necessários à arquitetura de auto-adaptação, pois possibilitam a introspecção e adaptação de *bindings* e a interceptação de mensagens para o monitoramento de QoS nas interfaces. Os meta-componentes são criados sob demanda por meio de uma operação oferecida pela cápsula, que recebe como parâmetro o nome único de um objeto (componente, interface ou *binding*) e retorna o meta-componente associado a esse objeto.

Um meta-componente pode ser criado apenas no contexto da cápsula que contém o objeto reificado. A cápsula utiliza o componente *BasicMOP* para a criação dos meta-componentes. Esse componente verifica se o objeto reificado é local ou não. Caso seja local, cria o meta-componente localmente, caso contrário, delega a criação para o componente *BasicMOP* da cápsula remota que contém o objeto. O componente *BasicMOP* retorna o nome único da interface do meta-componente recém criado. O nome é consultado pela cápsula no serviço de nomes, resultando em uma interface local ou uma referência para a interface remota do meta-componente.

A implementação dos meta-componentes de arquitetura e encapsulamento seguiu a mesma estratégia adotada nos protótipos anteriores e, deste modo, não é abordada em maiores detalhes nesta dissertação. De maneira geral, a estratégia consiste em combinar a meta-informação contida nos tipos com informações obtidas dos objetos reificados em tempo de execução, de modo a criar uma auto-representação desses objetos. Essa auto-representação pode então ser utilizada na introspecção e adaptação dinâmica dos objetos do nível base.

Já o meta-componente de interceptação foi inserido no protótipo MetaORB.NET para oferecer suporte a uma possível implementação das estratégias de monitoramento de QoS, como discutido no Capítulo 3, Seção 3.2, não tendo sido baseado em implementações de protótipos anteriores. Esse meta-componente pode ser criado apenas para interfaces. Através desse meta-componente, é possível inserir objetos interceptadores em uma interface, os quais podem adicionar pré- ou pós-processamento das mensagens que passam pela interface.

O Código 4.6 mostra a criação de um meta-componente de interceptação por meio do método *getInterceptMetaComp()* oferecido pela cápsula, usando o nome único de uma interface como parâmetro. O código mostra ainda a utilização do meta-componente para inserir um interceptador na interface. O interceptador utilizado é um interceptador para monitoramento de atraso (*DelayMonitorInterceptor*), que recebe como parâmetro a definição de um atributo de QoS referente ao atraso máximo aceitável em uma interação restrita por QoS. Esse código é semelhante ao utilizado pelo monitor de QoS.

**Código 4.6** Insersão de um interceptador em uma interface.

```
1.var metaComp = capsule.getInterceptMetaComp(interf.Uname);
2.metaComp.addInterceptor("delayInterceptor",
3.    new DelayMonitorInterceptor(attribute));
```

O Código 4.7 mostra, de maneira simplificada, a implementação do interceptador inserido no Código 4.6. A classe *DelayMonitorInterceptor* herda da classe *InterceptorBase*, que é a classe base para todos objetos interceptadores. Os interceptadores devem implementar os métodos virtuais *interceptMessageIn* e *interceptMessageOut* da superclasse, que contêm, respectivamente, o código de interceptação das mensagens que chegam à interface e das que deixam a interface.

**Código 4.7** Implementação de um interceptador.

```
1.public class DelayMonitorInterceptor : InterceptorBase
2.{
3.    public DelayMonitorInterceptor(XElement attribute){...}
4.
5.    protected override object[] interceptMessageIn(
6.        object impl, string methodName, object[] args){...}
7.
8.    protected override object[] interceptMessageOut(
9.        object impl, string methodName, object[] args){...}
  }
```

Não existe limite para o número de interceptadores que podem ser inseridos em uma interface. Os interceptadores inseridos são organizados em uma lista, de acordo com a ordem em que foram adicionados à interface. Deste modo, as mensagens que chegam ou deixam a interface atravessam essa lista, sendo processadas por cada um dos interceptadores.

## 4.3 Infra-estrutura de auto-adaptação

Como visto no Capítulo 3, a arquitetura da plataforma Meta-ORB foi estendida com um mecanismo para auto-adaptação. Esse mecanismo é formado por dois componentes principais, o gerenciador de adaptação e o monitor de QoS. Esses componentes foram definidos como tipos, usando a ferramenta gráfica apresentada na Seção 4.1, e armazenados no repositório de tipos. A Figura 4.7 mostra a definição dos componentes primitivos de gerenciamento de adaptação (*AMComp*) e monitoramento de QoS (*QoSMonitorComp*), juntamente com suas interfaces, definidos respectivamente nos módulos *SimpleAM* e *SimpleQoSMonitor*.

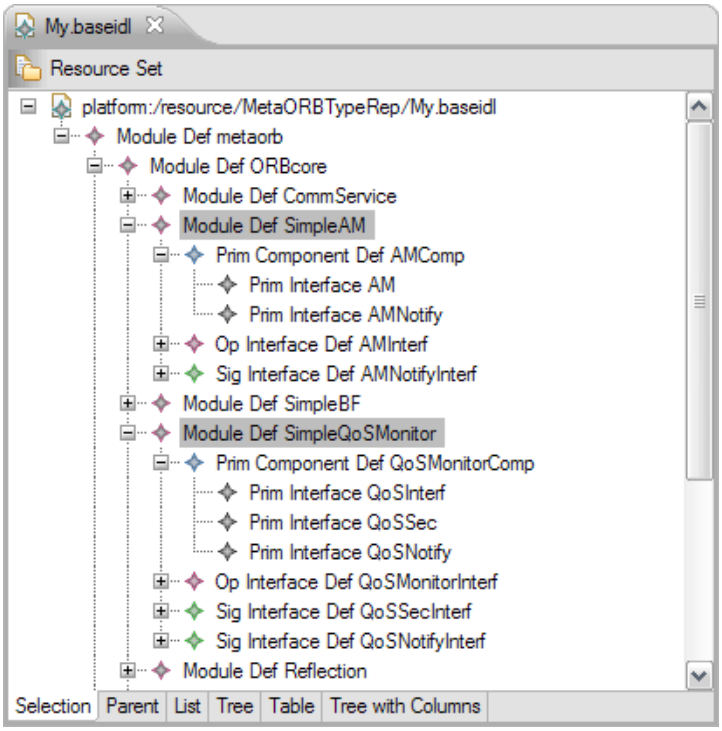



**Figura 4.7:** *Definição dos componentes de gerenciamento de adaptação e monitoramento de QoS.*

Após a definição no repositório de tipos, os componentes e suas interfaces foram implementados na linguagem C#, de acordo com as funcionalidades especificadas no Capítulo 3. A implementação das interfaces seguiu o padrão mostrado no Código 4.4, que mostra a implementação concreta da interface *AMInterf* do gerenciador de adaptação. A implementação dos componentes primitivos consistiu na criação de classes em C# contendo o código de cada uma das operações definidas nas interfaces, de acordo com as funcionalidades especificadas no Capítulo 3. O Código 4.8 mostra de maneira abstrata a implementação do componente primitivo *AMComp*.

**Código 4.8** Implementação do componente de gerenciamento de adaptação.

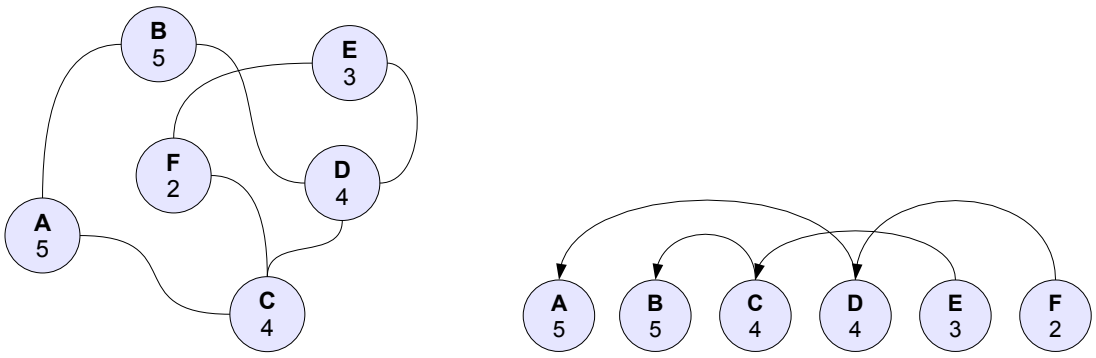
```
1.namespace MetaORB.NET.Implementations.OrbCore.SimpleAM{
2.    public class AdaptationManager : CBase, IAMInterf, IAMNotifyInterf
3.    {
4.
5.        public void register(string bindingName, string[] policiesIds)
6.        {
7.            // Implementação da operação register da interface AMInterf.
8.        }
9.
10.       public void notifyViolation(string interfUname,
11.           string attributeName, string measuredValue)
12.       {
13.           // Implementação da operação notifyViolation
14.           // da interface AMNotifyInterf.
15.        }
16.    }
17.}
```

A cápsula do protótipo MetaORB.NET foi então modificada para instanciar e oferecer acesso ao gerenciador de adaptação e ao monitor de QoS. O acesso é feito pelas interfaces locais *AMInterf* do gerenciador de adaptação e *QoSMonitorInterf* do monitor de QoS. Existe apenas uma única instância para cada um desses componentes, que são criados pela cápsula no primeiro acesso. Com os componentes definidos, implementados e acessíveis através da cápsula, a fábrica de *bindings* foi alterada para identificar políticas de adaptação associadas a um *binding* explícito em criação e registrá-las no gerenciador de adaptação, com a operação *register*.

A função da operação *register* é obter a definição dos tipos das políticas associadas ao *binding*, selecionar as que não sejam conflitantes e iniciar o monitoramento de QoS. O mecanismo de checagem de conflitos foi implementado seguindo as restrições discutidas no Capítulo 3, de modo que o gerenciador escolhe o maior conjunto de políticas não conflitantes, obedecendo sua prioridade. O algoritmo para checar conflitos monta um grafo com as políticas conflitantes, como o mostrado na Figura 4.8(a), onde os vértices representam as políticas (nomeadas de $A$ a $F$) e suas prioridades e as arestas indicam conflitos. A partir desse grafo, o grafo de políticas compatíveis, mostrado na Figura 4.8(b) é derivado.

Para criar o grafo da Figura 4.8(b), as políticas são inicialmente ordenadas por prioridade, da mais alta para a mais baixa. Depois, obedecendo essa ordem, cada uma das políticas é avaliada para derivar os caminhos mostrados no grafo, que representam os grupos de políticas compatíveis. No exemplo da Figura 4.8(a), os grupos derivados foram $\{A,D,F\}$ e $\{B,C,E\}$. Como existem dois grupos, o grupo contendo políticas de maior prioridade deve ser selecionado. Neste caso, o grupo $\{B,C,E\}$ é escolhido, pois as prioridades dessas políticas $(5,4,3)$ são maiores que as do outro grupo $(5,4,2)$.

Nesta implementação, o gerenciador avalia e aplica apenas o grupo de políticas escolhidas e as demais ficam em uma lista de espera, para uso somente no caso das pri-




(a) *Grafo de políticas conflitantes.* (b) *Grafo de políticas compatíveis.*

**Figura 4.8:** *Grafos que representam as políticas conflitantes (a) e compatíveis (b), gerados pelo algoritmo de checagem de conflitos.*

meiras falharem. O objetivo é simplificar o tratamento de conflitos para, na ocorrência da violação de um atributo de QoS, avaliar apenas políticas compatíveis. Outras implementações podem aplicar metodologias diferentes para a escolha das políticas, como por exemplo aplicar as políticas de acordo com a ordem em que ocorrem as violações de QoS. Nesse caso, mesmo uma política de prioridade mais baixa e conflitante com as demais poderia ser aplicada, caso a violação do atributo de QoS tratado por essa política seja a primeira a ocorrer.

Além da operação *register*, a implementação do componente primitivo *AMComp* conta com o código da operação *notifyViolation*, oferecida pela interface *AMNotifyInterf*. Esta operação é usada pelo monitor de QoS para notificar ao gerenciador de adaptação a violação de um atributo de QoS. Quando notificado, o gerenciador procura a política associada ao atributo violado na lista de políticas compatíveis e procede da maneira discutida no Capítulo 3.

Assim como o gerenciador de adaptação, o componente primitivo de monitoramento de QoS foi implementado de acordo com a arquitetura proposta no Capítulo 3. O monitor de QoS utiliza a abordagem de monitoramento baseada em interceptadores, empregando a implementação do meta-componente de interceptação discutida na Seção 4.2.3. Nessa abordagem, interceptadores com nomes derivados dos nomes dos atributos de QoS são instanciados pelo monitor e inseridos nas interfaces monitoradas.

A implementação baseada em interceptadores é uma alternativa simples para monitorar interfaces com restrições de QoS. Entretanto, os interceptadores são artefatos primitivos e dependentes de linguagem de programação e não são descritos como tipos no repositório. Desta forma, para usá-los efetivamente como monitores de contexto reaproveitáveis, a implementação carece ainda de um mecanismo para o gerenciamento adequado desses artefatos, como aquele existente para os tipos. Trabalhos futuros podem

estudar a substituição dessa abordagem por um mecanismo de provisão de contexto mais completo.

## 4.4 Considerações Finais e Trabalhos Futuros

Este capítulo apresentou o protótipo da plataforma Meta-ORB desenvolvido para validar a arquitetura proposta no Capítulo 3, chamado de MetaORB.NET. Esse protótipo conta apenas com a infra-estrutura necessária para implementação da arquitetura de auto-adaptação e para sua avaliação em um estudo de caso, apresentado no Capítulo 5. As funcionalidades desenvolvidas e sua relação com as extensões para auto-adaptação são listadas abaixo:

- Sistema de tipos integrado com o protótipo - Possibilita o gerenciamento global dos tipos que descrevem tanto as configurações do middleware e suas aplicações quanto as políticas de adaptação envolvidas em tais aplicações;
- Modelo de programação implementado em C# - Suporte à construção da infra-estrutura de auto-adaptação (gerenciador de adaptação e monitor de QoS), bem como das aplicações auto-adaptativas, de acordo com o modelo de componentes da plataforma Meta-ORB;
- Meta-nível - Suporte à adaptação dinâmica do middleware, que é a base do protocolo de auto-adaptação realizado pelo gerenciador de adaptação. Além disso, as estratégias de monitoramento de QoS implementadas foram baseadas no mecanismo reflexivo de interceptação.

O protótipo MetaORB.NET, apesar de ser funcional, tem por objetivo apenas demonstrar a viabilidade da arquitetura de auto-adaptação. Neste sentido, o protótipo pode ser empregado experimentalmente, apesar de não contar com todas as funcionalidades necessárias em alguns cenários reais. Existem ainda muitas funcionalidades e melhorias que precisam ser estudadas e implementadas, tendo em vista que uma das propostas do trabalho apresentado nesta dissertação é apoiar o desenvolvimento e gerenciamento de aplicações auto-adaptativas.

Algumas funcionalidades do middleware, como o suporte a componentes compostos e a implementação dos demais meta-níveis, ainda precisam ser inseridas no protótipo. Além da infra-estrutura básica do middleware, o desenvolvedor pode se beneficiar de ferramentas que auxiliem o desenvolvimento das aplicações. A ferramenta gráfica para a definição de tipos é um exemplo, apesar de ainda ser muito restrita. Essa ferramenta pode ser aprimorada com uma interface gráfica mais rica para a modelagem dos tipos, assim como as ferramentas convencionais para modelagem UML. Além disso, o serviço remoto

do repositório possibilita apenas a consulta, sendo que o suporte à definição remota de tipos ainda precisa ser implementado.

Atualmente, a instanciação de um tipo dependente de código de implementação, como no caso de componentes e *bindings* primitivos, só é possível se esse código estiver disponível entre as classes carregadas durante a inicialização do sistema. Outra melhoria que pode ser adicionada ao repositório de tipos é o suporte a *templates* que contenham o código de implementação de um tipo escrito com uma linguagem de programação específica. Deste modo, durante a instanciação do tipo, seu código de implementação pode ser baixado do repositório e carregado dinamicamente.

Uma outra ferramenta que pode auxiliar o desenvolvimento de aplicações é um gerador de código, capaz de interpretar o tipo de uma interface Meta-ORB e gerar a implementação correspondente dessa interface em uma linguagem de programação. Essa ferramenta pode, por exemplo, ler o tipo mostrado no Código 4.3 e gerar a implementação mostrada no Código 4.4. Isso é possível pois, como visto na Seção 4.2.2, a meta-informação contida na definição do tipo de uma interface é suficiente para construir sua implementação. Além disso, a ferramenta pode gerar classes base para a implementação de componentes primitivos, contendo a assinatura dos métodos que devem ser implementados pelo desenvolvedor.

CAPÍTULO 5

# Estudo de Caso

Este capítulo apresenta um estudo de caso realizado para avaliar os benefícios da arquitetura de middleware proposta no desenvolvimento de uma aplicação auto-adaptativa. A aplicação escolhida foi um quadro branco compartilhado, que consiste em uma aplicação colaborativa onde os usuários podem interagir uns com os outros usando tinta digital.

O Instituto de Informática possui um laboratório formado por tablets PCs, adquiridos através do programa *HP Technology for Teaching*, onde é desenvolvido um projeto educacional para aplicar a computação móvel e a tinta digital em disciplinas do curso de Ciência da Computação. A idéia de desenvolver uma aplicação como o quadro branco surgiu desse projeto, com o propósito de oferecer uma área compartilhada de anotações para professores e alunos, que podem fazer contribuições em tempo real, através de seus tablets PCs, usando a tinta digital. Essa aplicação foi projetada para utilização em ambientes móveis e, desta forma, considera aspectos de qualidade de serviço envolvidos na comunicação em tempo real da tinta digital nesse tipo de ambiente.

O estudo de caso apresentado neste capítulo envolveu as seguintes etapas:

- Estudo sobre tinta digital para conhecer a tecnologia e sua aplicabilidade, discutido brevemente na Seção 5.1;
- Definição da aplicação (quadro branco) e desenvolvimento de um protótipo preliminar para testar suas propriedades. Os testes mostraram que problemas típicos do ambiente móvel, como o atraso e a perda de pacotes, podem afetar a aplicação. Deste modo, foram realizados experimentos para avaliar o efeito desses problemas na comunicação da tinta digital. Essa etapa é apresentada na Seção 5.2;
- Estudo preliminar sobre a caracterização da tinta digital como um tipo de mídia. O estudo mostra que existe a necessidade de tratar adequadamente a comunicação em aplicações como o quadro branco para manter o nível de qualidade de serviço esperado pelo usuário. Esse estudo é apresentado na Seção 5.3;
- Desenvolvimento de um quadro branco auto-adaptativo usando a plataforma MetaORB.NET. Com isso, a aplicação se torna capaz de identificar mudanças no ambiente (relativas ao aumento do atraso e das perdas de pacotes) e se adaptar auto-

maticamente, tentando manter o nível de QoS esperado pelo usuário. Essa etapa é apresentada na Seção 5.4;

- Experimentos com o propósito de avaliar o impacto da adaptação na comunicação da tinta digital. Essa avaliação é discutida na Seção 5.5, seguida de considerações gerais sobre o estudo de caso, apresentadas na Seção 5.6.

## 5.1 Tinta Digital

O desenho e a escrita são atividades realizadas naturalmente pelas pessoas para organizar idéias sem se preocupar com a precisão com que as mesmas são representadas [29]. Modelos podem ser desenhados para esquematizar objetos reais, conceitos podem ser escritos para memorização e associações entre idéias podem ser facilmente visualizadas quando desenhadas.

Com o intuito de tornar a interação humano-computador mais natural, surgiram os dispositivos computacionais equipados com caneta e as aplicações baseadas em tinta digital. Computadores portáteis, conhecidos como *Tablet PCs*, são equipados com um hardware especial, constituído de uma caneta e um digitalizador em sua tela. O movimento e o contato da caneta com a superfície da tela é capturado pelo digitalizador, formando o que se convencionou chamar de tinta digital, que consiste, essencialmente, na representação dos traços realizados com a caneta. Além disso, esses dispositivos contam com as vantagens oferecidas pelo poder computacional, mobilidade e conectividade equivalentes aos dos *laptops*.

O digitalizador coleta informações sobre as coordenadas da ponta da caneta durante sua movimentação, bem como a pressão no ponto de contato da caneta com a superfície da tela. Essas informações consistem em seqüências numéricas que são armazenadas em estruturas conhecidas como pacotes de tinta. Os pacotes são analisados por software, que renderiza a imagem desenhada com a caneta na tela do tablet PC em tempo real.

A tinta digital é formada por estruturas que representam riscos feitos com a caneta, ou *strokes*, que contêm os pacotes de tinta coletados durante cada contato da caneta com a superfície da tela. Os riscos que formam a tinta digital podem ser interpretados pelas aplicações de diversas maneiras, como, por exemplo, no reconhecimento de escrita e de formas geométricas, onde a tinta coletada é transformada em texto ou em objetos gráficos. Até mesmo a seqüência de gestos com a caneta pode ser convertida em dados de entrada para as aplicações.

Pesquisas sobre tinta digital vêm sendo realizadas há algum tempo, como em [47], que descreve uma ferramenta de busca de padrões em tinta digital e em [13], que trata da utilização da tinta em conjunto com vídeo em uma sala de conferência. Este último

explora aspectos interativos da tinta, utilizada como uma ferramenta simples para tomar notas durante uma conferência. Pesquisas atuais exploram outros aspectos da tinta digital, como o aspecto temporal [54], que define o momento e a ordem em que cada anotação com a caneta foi realizada. Em [11] é proposto um *player* para aprimorar a visualização de anotações de tinta digital sobre documentos, onde a tinta armazenada pode ser revista de maneira interativa.

## 5.2 Quadro Branco Compartilhado

A tinta digital proporciona uma forma de tomar notas próxima daquela usada comummente pelas pessoas. Assim, os tablet PCs podem ser usados de maneira mais intuitiva que computadores com dispositivos de interação convencionais (mouse e teclado). Diversas aplicações podem tirar proveito dessa tecnologia, como, por exemplo, aquelas destinadas ao ensino.

Aplicações computacionais já utilizadas em sala de aula, como recursos áudio-visuais, podem se tornar interativas através da tinta digital. A ferramenta Classroom Presenter [1] é um exemplo. Com o uso dessa ferramenta, o modelo tradicional de apresentação de *slides* é aprimorado com a possibilidade de anotações em tempo real sobre a apresentação, tanto por parte dos alunos como por parte do professor.

O quadro branco compartilhado é outro exemplo de aplicação da tinta digital e é o foco do estudo de caso apresentado nesta dissertação. O quadro branco pode ser utilizado tanto em sala de aula quanto em outras atividades onde é importante uma área compartilhada para anotações. A idéia da utilização do quadro branco como recurso computacional não é nova [5]. Porém, este estudo de caso visa explorar os aspectos de QoS envolvidos na comunicação da tinta digital que podem afetar um sistema dessa natureza.

Em comparação com um quadro branco real, o quadro branco computacional apresenta algumas vantagens, como a manipulação direta da tinta para operações de escala, recorte, movimentação e alteração de propriedades como cor e estilo dos traços, além da possibilidade de gravação da tinta para consulta futura. Nesta aplicação, um usuário pode dar início a uma seção escolhendo os outros usuários que farão parte de um grupo colaborativo. Em uma sala de aula, por exemplo, o professor pode formar grupos de alunos que vão trabalhar em conjunto para resolver um exercício proposto. Os alunos do grupo podem fazer contribuições no quadro branco, que são replicadas para os demais alunos em tempo real. Além disso, os alunos podem configurar a cor, espessura do traço e transparência, entre outras propriedades da tinta.

Um protótipo inicial do quadro branco foi desenvolvido para testar as propriedades desse tipo de aplicação e é mostrado na Figura 5.1. Esse protótipo foi desenvolvido

sem o auxílio de uma infra-estrutura de middleware. Assim, a tinta é transferida por cada usuário para os demais à medida em que é coletada, sem nenhum tratamento, por uma rede sem fio. Sem um tratamento adequado, problemas típicos de rede, tais como atrasos e perdas de pacotes, podem causar falhas na coordenação de atividades, na representação da tinta e, conseqüentemente, na colaboração.

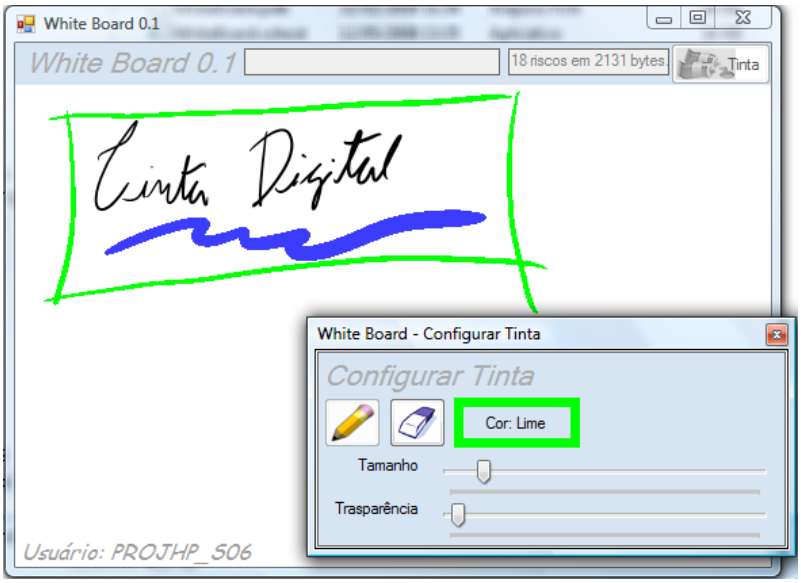



**Figura 5.1:** *Quadro branco compartilhado: primeiro protótipo.*

### 5.2.1 Problemas que Afetam a Interação

O ambiente de rede sem fio, apesar dos benefícios oferecidos pela mobilidade, possui uma série de limitações em comparação com as redes cabeadas. Largura de banda menor, maior taxa de erros e desconexão abrupta são alguns dos problemas encontrados nesse tipo de ambiente [26]. Esses problemas tendem a afetar a comunicação em aplicações distribuídas, pois causam perdas e atrasos dos pacotes enviados pela rede.

Alguns experimentos foram realizados para avaliar o impacto do atraso e da perda de pacotes em aplicações baseadas na comunicação em tempo real da tinta digital. Os experimentos foram conduzidos no laboratório de tablet PCs do Instituto de Informática enquanto alunos faziam uso normal dos tablet PCs e da rede. O laboratório possui vinte tablet PCs (modelo HP Compaq 2710p, Intel Core 2 Duo 1.2$GHz$, 2$GB$ RAM) conectados por uma rede sem fio 802.11$a$.

Uma aplicação de monitoramento foi desenvolvida para gerar fluxos de pacotes entre dois pontos da rede e coletar informações sobre atraso e perda de pacotes. Os pacotes utilizados possuíam tamanhos similares ao tamanho típico dos pacotes de tinta digital gerados pelo quadro branco compartilhado. A Figura 5.2 mostra um gráfico com os dados coletados durante o período de quarenta minutos em uma aula da disciplina

de Programação de Computadores, realizada no primeiro semestre do ano de 2008. No gráfico, os pontos representam o atraso médio (*Delay*) e a porcentagem de perdas (*Loss*), calculados a cada intervalo de 100 pacotes.

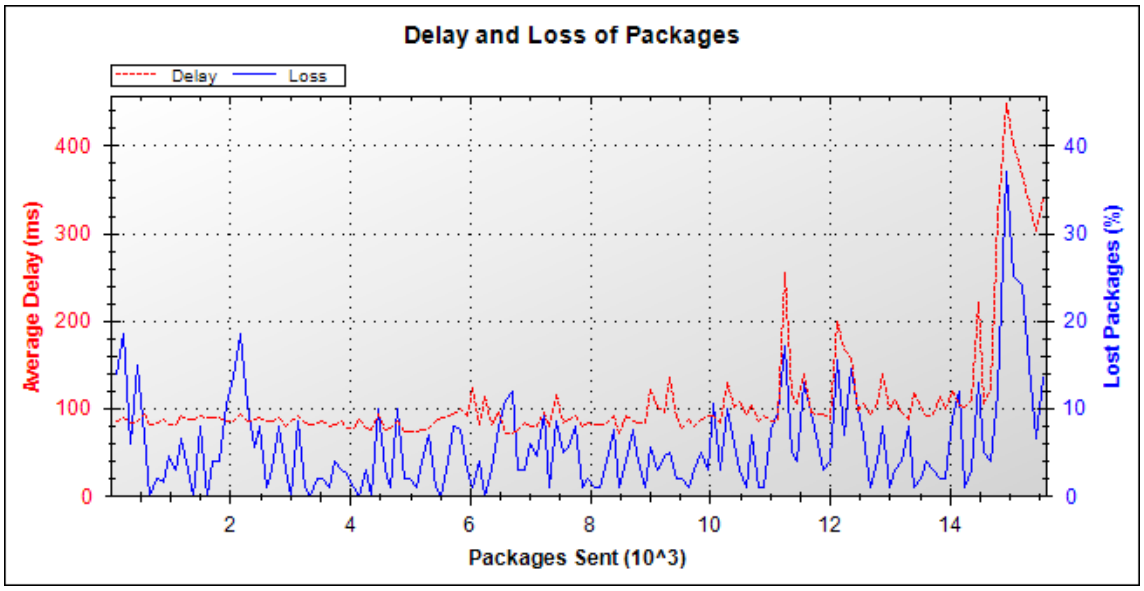



**Figura 5.2:** *Dados de atraso e perdas coletados durante uma aula em laboratório*

Como pode ser visto no gráfico, próximo ao fim do experimento, houve uma mudança repentina nos valores monitorados, com um aumento considerável no atraso médio e na porcentagem de perdas. Isto mostra que, em ambientes móveis reais, os valores observados de atraso e perdas não se mantêm constantes durante a execução de uma aplicações como o quadro branco. Vários fatores podem afetar esses valores, como a mudança dinâmica do número de nós conectados na rede, bem como detalhes da implementação do ponto de acesso e das placas de rede presentes nos dispositivos.

Os dados coletados no experimento foram usados para configurar uma aplicação de simulação, desenvolvida para verificar os efeitos do atraso e da perda de pacotes na comunicação da tinta digital. Essa aplicação funciona em um único tablet PC, sem a necessidade de um ambiente de rede e recebe como parâmetro o atraso médio e a freqüência perda de pacotes. Esses parâmetros são aplicados na tinta digital coletada em um painel da aplicação e os efeitos podem ser vistos simultaneamente em outro painel.

O problema do atraso afeta diretamente a coordenação das atividades. No quadro branco compartilhado, por exemplo, o professor pode estar falando enquanto escreve no quadro. Se houver atraso na recepção da tinta pelos alunos, aquilo que o professor fala pode parecer fora de sincronia com o que ele escreve. A Figura 5.3 mostra uma simulação realizada com os dados de atraso obtidos nos experimentos. Considerando o cenário do quadro branco em sala de aula, a figura do painel esquerdo seria a visão do professor e a do painel direito a do aluno em um mesmo momento. Em decorrência do atraso, o aluno

pode estar ouvindo uma explicação sobre algo que ainda não aparece em sua cópia do quadro branco.


**Figura 5.3:** *Atraso na recepção de pacotes de tinta digital*

Já as perdas de pacotes afetam a representação estática da tinta em um determinado momento. Como os pacotes de tinta consistem em dados de coordenadas, os trechos de tinta formados por essas coordenadas são perdidos. Por exemplo, se houver perdas de pacotes contendo determinados trechos de palavras, uma frase escrita no quadro branco pelo professor pode se tornar incompreensível para os alunos. A Figura 5.4 ilustra essa situação através da simulação feita com os dados colhidos no experimento. A figura do painel da esquerda mostra a tinta produzida pelo professor e a do painel da direita mostra como essa tinta seria visualizada pelos alunos, com perdas que podem comprometer seu entendimento.

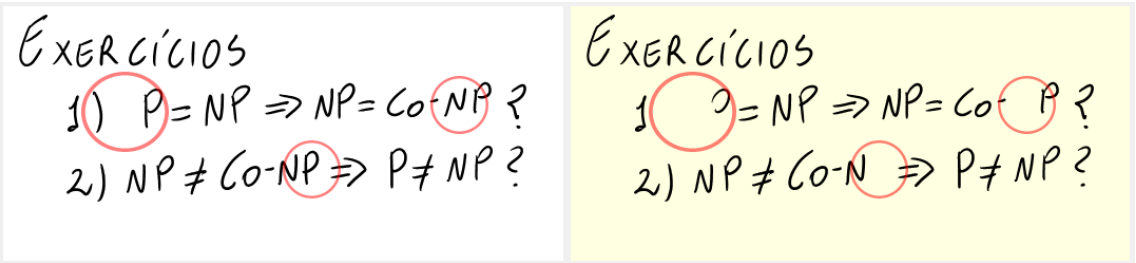



**Figura 5.4:** *Efeito da perda de pacotes de tinta digital.*

## 5.3 Tinta Digital: Um Novo Tipo de Mídia

Como visto na seção anterior, em aplicações como o quadro branco compartilhado, a tinta digital deve ser transmitida em tempo real entre os usuários da aplicação. Com os experimentos, foi observado que aplicações de tempo real que envolvem a comunicação de tinta digital apresentam uma certa tolerância a atrasos e perdas. Atrasos mínimos em sua recepção, assim como pequenas perdas, podem não ser percebidos pelos usuários. Porém, quando o atraso e as perdas aumentam, o usuário perde a sensação de recepção em tempo real da tinta e não consegue entender seu conteúdo. Essa sensibilidade

ao atraso e a perdas de pacotes é característica das aplicações de multimídia distribuída, como as que envolvem áudio e vídeo em tempo real.

Um outro aspecto importante da comunicação a ser considerado é o formato dos dados que serão transmitidos. Para possibilitar a interoperabilidade entre aplicações com finalidades distintas, a tinta digital deve ser armazenada ou distribuída em um formato próprio, independente da aplicação. Em [3] a tinta digital é considerada um tipo de dado de primeira classe. Assim, aplicações distintas podem utilizar o mesmo formato para representá-la e, desta forma, interoperar.

Da mesma forma, em aplicações distribuídas, como o quadro branco mostrado na Figura 5.1, a tinta digital deve seguir um formato padrão que possibilite sua transmissão em tempo real através de um fluxo de dados contínuo. Formatos proprietários, como o ISF (*Ink Serialized Format*) da Microsoft [65], são inadequados pois não permitem a manipulação direta do conteúdo da tinta para realizar o controle do fluxo. Já formatos abertos, como o InkML (*Ink Markup Language*) [58] são mais adequados, pois definem uma linguagem padrão para o compartilhamento da tinta entre dispositivos e plataformas heterogêneos. Tanto ISF quanto InkML utilizam basicamente a mesma informação para representar a tinta (dados de coordenada e pressão). Assim, aplicações que utilizam formatos distintos podem interoperar, desde que haja a conversão dos pacotes de tinta de um formato para o outro.

Como a tinta digital pode ser considerada um tipo de dado de primeira classe (independente de linguagem e de aplicação) e compartilha características com tipos de mídia como áudio e vídeo, então é natural tratá-la como um novo tipo de mídia em aplicações distribuídas [37]. Portanto, a comunicação em aplicações que envolvem a transmissão da tinta digital em tempo real deve receber o mesmo tratamento que outras aplicações multimídia.

O tratamento da comunicação em aplicações de multimídia distribuída envolve a manutenção da qualidade de serviço oferecida ao usuário. Em aplicações de áudio e vídeo, por exemplo, caso a largura de banda não seja suficiente para a transmissão em tempo real, então são aplicadas técnicas de compressão de áudio e vídeo para diminuir a largura de banda necessária. Técnicas de compressão também podem ser aplicadas à tinta digital. Um exemplo é a diminuição do número de pontos de coordenada que formam um *stroke*. Aplicando essa compressão, os traços que formam a tinta perdem um pouco de precisão, porém o volume de dados trafegados pela rede diminui. A Figura 5.5 ilustra a compressão realizada por meio da diminuição do número de pontos, onde os pontos que formam a imagem do painel esquerdo foram reduzidos pela metade para formar a imagem do painel direito. Apesar de diminuir a qualidade da imagem, o conteúdo da tinta digital ainda pode ser entendido pelo usuário.

Levando em consideração o desenvolvimento de aplicações baseadas em tinta

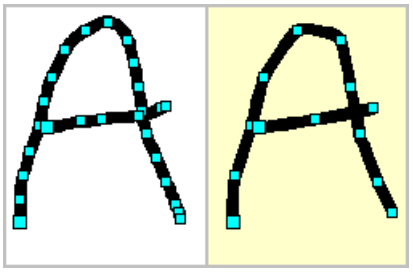


**Figura 5.5:** *Compressão da tinta digital.*

digital que utilizam a plataforma Meta-ORB, caracterizar a tinta digital como um tipo de mídia implica em sua inclusão no sistema de tipos da plataforma. O meta-modelo da plataforma, como discutido no Capítulo 2, oferece construções para a especificação dos tipos de mídia envolvidos em interações realizadas entre interfaces de fluxo contínuo (*StrInterfaceDef*). Como visto anteriormente na Figura 2.9, cada um dos fluxos (*FlowDef*) de uma interface está associado a uma especificação de mídia (*MediaSpecificationDef*), que define um conjunto de tipos de mídia específicos para os quais o fluxo provê suporte.

O tipo de mídia usado depende de um processo de negociação, onde os envolvidos devem entrar em acordo sobre o tipo de mídia empregado na interação. Cada tipo de mídia específico fornece a meta-informação necessária para qualificar a mídia em um ambiente distribuído. A tinta digital, por exemplo, possui uma série de atributos que podem ser usados para qualificá-la, como o formato dos traços, que define o formato utilizado para codificar um traço ou risco [58]. O formato mais simples é constituído de dois canais, ou duas dimensões, correspondentes às coordenadas $X$ e $Y$. Nesse formato ($XY$) um traço é codificado como uma seqüência de valores para as coordenadas $X$ e $Y$. Tablet PCs convencionais possuem ainda o canal $F$ correspondente à força ou pressão empregada pela ponta da caneta na superfície da tela. Nesses dispositivos, o formato usado ($XYF$) é codificado como uma seqüência de valores para as coordenadas $X$ e $Y$ de um ponto mais a pressão naquele ponto.

Na negociação do tipo de mídia, se algum dos nós envolvidos for um dispositivo que não suporta o canal de pressão ($F$), então o formato utilizado deve ser o $XY$. Por outro lado, esse dispositivo pode gerar valores nulos para esse canal e deste modo aceitar o formato $XYF$. O formato dos traços é apenas um dos atributos que podem ser empregados na negociação. Existem ainda outros atributos, como a taxa de amostragem, que define a taxa com que o digitalizador gera amostras de tinta, e a área ativa, que define os limites da área bidimensional de escrita. Os valores desses atributos são determinados pelo tipo de dispositivo empregado (*laptop*, *tablet PC*, *pocket PC*, etc.) e são úteis para a negociação em interações que envolvem dispositivos heterogêneos. Outros tipos de atributos podem ainda ser usados como parâmetros para configurar inicialmente a interação. Um desses atributos, usado no tipo de mídia da tinta digital, é o intervalo de amostragem, que define

o intervalo em milissegundos de coleta de amostras de tinta.

Neste trabalho, um estudo preliminar da caraterização da tinta digital como mídia foi realizado, de modo que ainda é necessário um estudo completo desse tipo de mídia e seus atributos. Além disso, a implementação atual da plataforma não define um processo de negociação do tipo de mídia durante a criação do *binding*. Assim, apenas os atributos usados como parâmetros para a configuração das aplicações são considerados. O estudo desse tipo de mídia deve considerar ainda os parâmetros de QoS envolvidos em sua comunicação para determinar, de forma mais precisa, os níveis de QoS esperados pelo usuário de aplicações baseadas em tinta digital.

A Figura 5.6 mostra a definição da tinta digital como um tipo de mídia, feita com o auxílio da ferramenta gráfica do repositório de tipos. Primeiro, um tipo de mídia genérico (*GenericMediaTypeDef*) é criado para registar a tinta digital como um dos tipos de mídia suportado pela plataforma. Depois, é criada uma especificação de mídia (*MediaSpecificationDef*), que será associada aos fluxos de tinta digital (*InkFlowIn* e *InkFlowOut*). Essa especificação contém apenas um tipo de mídia específico (SpecificMediaTypeDef) que qualifica a tinta digital usada em tablet PCs (TabletPcInk), contendo os atributos formato do traço (*traceFormat*), taxa de amostragem (*sampleRate*) e intervalo de amostragem (*samplingInterval*). Como visto anteriormente, os dois primeiros atributos são úteis para a negociação do tipo de mídia, enquanto o terceiro pode ser empregado para configurar a aplicação.

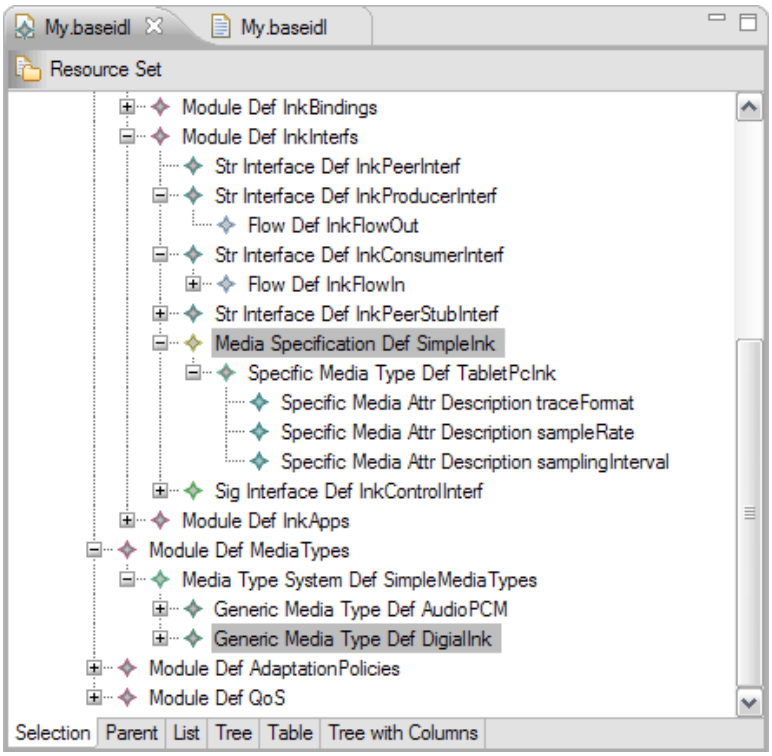



**Figura 5.6:** *Definição da tinta digital como um tipo de mídia no repositório de tipos.*

## 5.4 Quadro Branco Compartilhado Auto-Adaptativo

A Seção 5.2 apresentou a aplicação usada como estudo de caso, que é um quadro branco compartilhado baseado em tinta digital. Como discutido anteriormente, através do protótipo desenvolvido inicialmente e de experimentos realizados em uma rede sem fio, foram identificados problemas envolvendo a comunicação da tinta digital que podem afetar a interação entre usuários dessa aplicação. Em uma tentativa de contornar ou minimizar esses problemas, a aplicação precisa identificar mudanças em seu contexto de execução e realizar adaptações automaticamente. Tendo em vista essa necessidade, este estudo de caso visa empregar a plataforma MetaORB.NET para adicionar comportamento auto-adaptativo ao quadro branco.

Primeiramente, a aplicação precisa ser definida de acordo com o modelo de programação da plataforma Meta-ORB, ou seja, em termos de componentes, interfaces e *bindings*. Isso pode ser feito com a ferramenta gráfica para definição de tipos do repositório de tipos. A Figura 5.7 ilustra as entidades que formam a aplicação em tempo de execução, instanciadas a partir dos tipos que formam o modelo da aplicação. Os tipos, por sua vez, são construídos de acordo com o meta-modelo da plataforma.

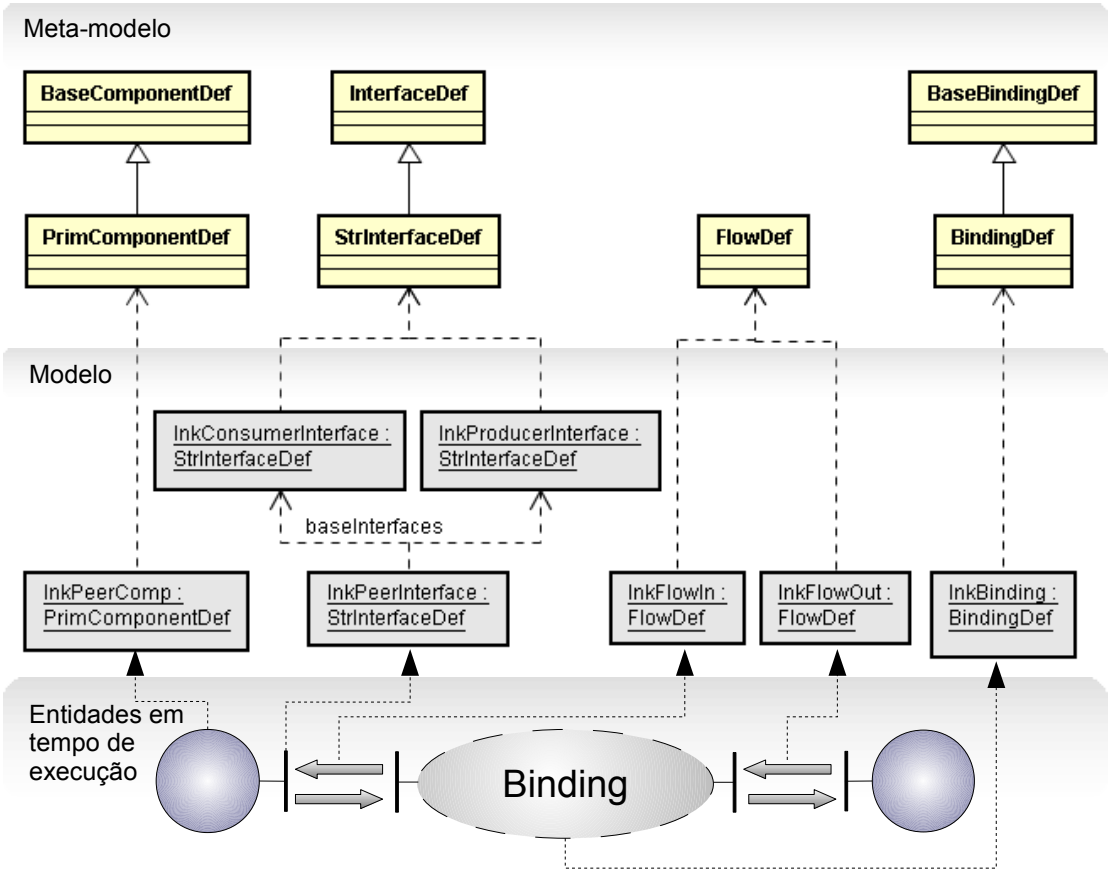



**Figura 5.7:** *Quadro Branco Compartilhado: Modelo da aplicação.*

Na Figura 5.7, as associações presentes no modelo e no meta-modelo foram excluídas para simplificar a visualização das entidades. O componente primitivo *InkPeerComp* possui apenas a interface de fluxo contínuo *InkPeerInterface*, definida em termos de duas interfaces base, *InkProducerInterface* e *InkConsumerInterface*. As interfaces base adicionam à interface *InkPeerInterface* os comportamentos de produtor e de consumidor de amostras de tinta digital, realizados através dos fluxos *InkFlowOut* e *InkFlowIn*, respectivamente. Assim, o componente primitivo *InkPeerComp* implementa as funcionalidades para produzir e enviar um fluxo contendo amostras de tinta geradas localmente e para consumir fluxos contendo amostras de tinta recebidas de produtores remotos.

Com os componentes para produção e consumo de tinta digital definidos, resta definir a configuração interna do *binding InkBinding*, mostrado na Figura 5.7. Esse *binding* define como é realizada a comunicação da tinta digital entre os componentes remotos. Em um modelo simples de comunicação, o *binding* pode ser formado apenas por um componente *stub*, responsável por serializar e despachar pacotes contendo amostras de tinta digital, conectado a um *binding* primitivo que implementa um protocolo de transporte do tipo *Multicast UDP*. A Figura 5.8 mostra a configuração interna do *binding InkBinding* instanciada a partir do tipo do *binding*, seguindo o modelo simplificado de comunicação. O tipo do *binding* define quais papéis podem ser realizados em cada um dos *endpoints* do *binding*, em termos de uma configuração interna de componentes e de uma interface alvo.

De acordo com a Figura 5.8, o *binding InkBinding* define apenas o papel *InkPeerRole*, cuja configuração é formada apenas pelo componente interno *InkPeerStub*. A interface alvo definida nesse papel é uma interface do componente *InkPeerStub* compatível com a interface *InkPeerInterface* do componente *InkPeerComp*. O tipo do *binding* define ainda seus *bindings* internos. O *binding InkBinding* possui apenas um *binding* interno, o *binding* primitivo *PBMulticastUDP*.

Com a configuração definida até o momento, é possível instanciar componentes do tipo *InkPeerComp* em diversas cápsulas remotas e criar um *binding* do tipo *InkPeerBinding* para conectar esses componentes, usando suas interfaces de fluxo contínuo do tipo *InkPeerInterface*. Entretanto, os requisitos de QoS para a comunicação da tinta não são considerados. Esses requisitos podem ser incluídos no modelo da aplicação como anotações de QoS associadas ao fluxo de tinta *InkFlowIn*, de acordo com o meta-modelo mostrado na Figura 2.9.

Como visto na Seção 5.2.1, os principais problemas que afetam a interação em uma aplicação como o quadro branco decorrem do atraso na recepção e da perda dos pacotes contendo amostras de tinta digital. Sendo assim, os requisitos de QoS dessa aplicação podem ser definidos em termos de atributos que especificam valores aceitáveis de atraso e perdas. A Figura 5.9 ilustra a anotação do fluxo *InkFlowIn* com tais atributos.

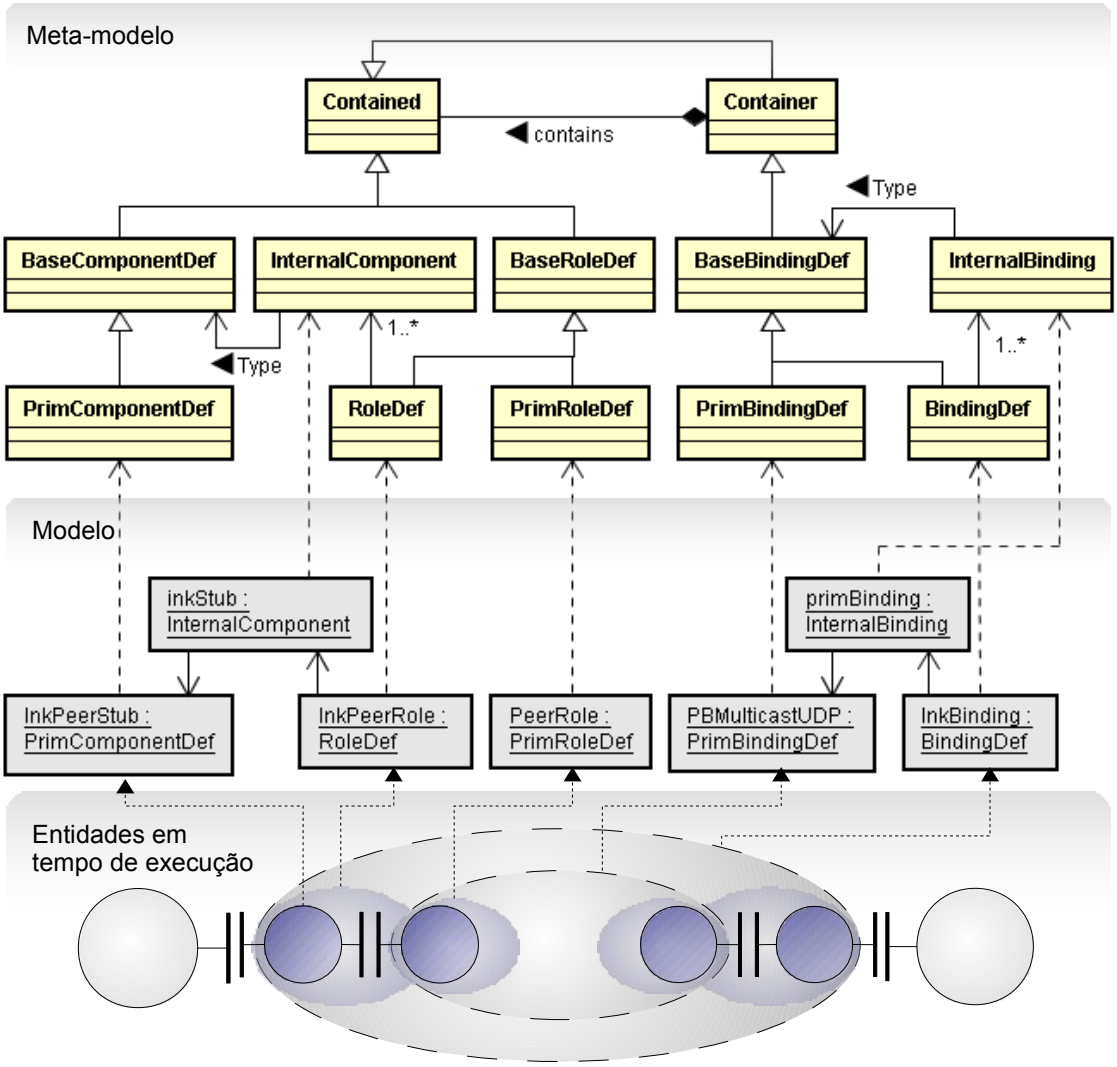



**Figura 5.8:** *Quadro Branco Compartilhado: Configuração interna do binding.*

Na figura, os atributos *DelayQoSAttr* e *LossQoSAttr* definem respectivamente o valor máximo aceitável para o atraso (600*ms*) e o valor máximo aceitável para as perdas (4%).

A meta-informação contida nos atributos de QoS é usada pelo monitor de QoS para determinar quando o gerenciador de adaptação deve ser notificado sobre uma violação, como visto no Capítulo 3. No caso dos atributos mostrados na Figura 5.9, o monitor de QoS irá notificar o gerenciador de adaptação sempre que o atraso na recepção dos pacotes ultrapassar 600*ms* e sempre que a porcentagem de perdas ultrapassar 4%.

Os valores de 600*ms* para o atributo de atraso e de 4% para o atributo de perdas foram estabelecidos para a aplicação com base em observações feitas durante os experimentos envolvendo a comunicação da tinta digital, mas não são definitivos. Um estudo mais aprofundado sobre a caracterização da tinta como um tipo de mídia precisa ser realizado para determinar de forma mais precisa o nível de QoS esperado pelos usuários, principalmente com relação aos parâmetros de atraso e perdas.

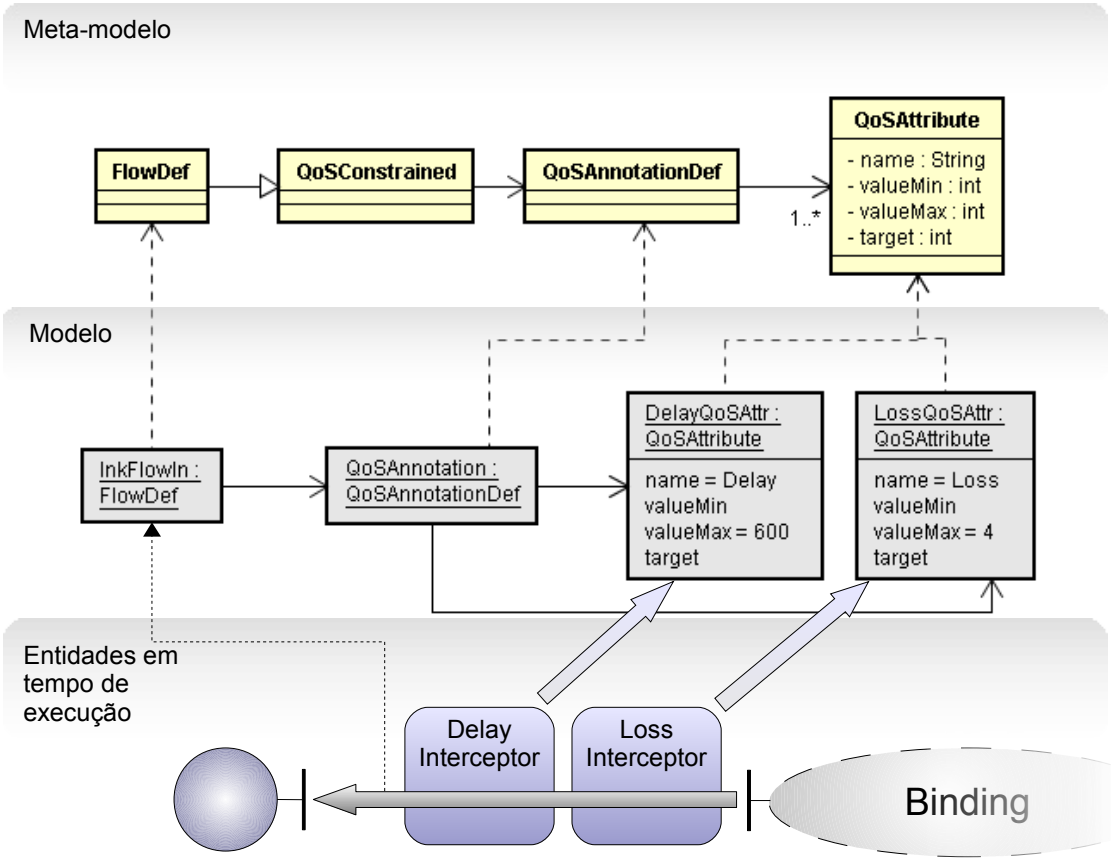



**Figura 5.9:** *Quadro Branco Compartilhado: Atributos de QoS.*

Por fim, é preciso definir as políticas de adaptação associadas ao *binding InkBinding*. A Figura 5.10 ilustra a definição de uma política de adaptação para tratar a violação do atributo de QoS referente às perdas. Quando notificado pelo monitor de QoS, o gerenciador de adaptação aplica a política referente ao atributo de QoS que gerou a violação. A política define regras contendo a informação necessária para a invocação de uma das operações do meta-componente de arquitetura que reifica o *binding*. No caso da política mostrada na Figura 5.10, a operação utilizada é *RoleReplaceComponent*, que substitui um dos componentes internos de um *binding* por outro em todos os *endpoints* do *binding*.

A adaptação dirigida pela política mostrada na Figura 5.10 consiste em substituir o componente *InkPeerStub* pelo componente *InkQoSStub* em todos os *endpoints* do *binding* que realizam o papel *InkPeerRole*. Como o *binding InkBinding* define apenas o papel *InkPeerRole*, essa operação afeta todos os *endpoints* do *binding* e, conseqüentemente, todos os usuários da aplicação. O componente *InkQoSStub* contém as mesmas interfaces que o componente *InkPeerStub* e passa a ser o *stub* ligado ao componente *InkPeerComp* e ao *binding* primitivo *PBMulticastUDP*. Esse componente, além de realizar o papel de *stub*, implementa um mecanismo de recuperação de pacotes de tinta perdidos, fazendo com que a porcentagem de perdas caia para 0%.

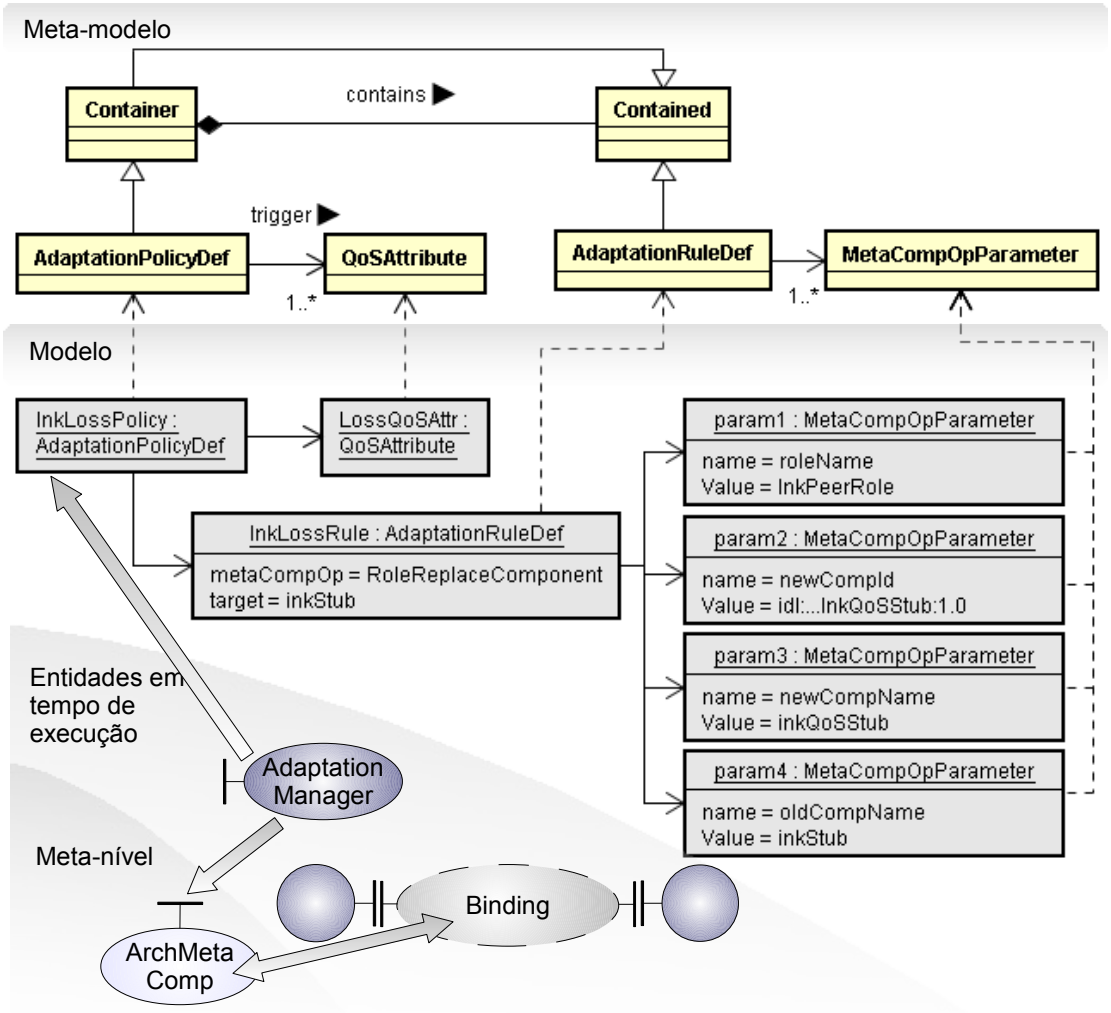



**Figura 5.10:** *Quadro Branco Compartilhado: Políticas de adaptação.*

A Figura 5.10 ilustra apenas uma política para tratar a violação do atributo *LossQoSAttr*. Além desse atributo, o fluxo *InkFlowIn* está associado ao atributo *DelayQoSAttr*, referente ao valor máximo de atraso tolerável. Da mesma forma, uma política poderia ser definida para tratar esse atributo, usando, por exemplo, um componente que introduz a compressão da tinta digital, usando o método mostrado na Figura 5.5. Entretanto, diferente do mecanismo para recuperação de erros, a compressão dos dados não garante a diminuição no atraso para um nível tolerável. O objetivo da política é tentar melhorar a interação entre os usuários diminuindo a largura de banda usada pela aplicação.

Com as políticas de adaptação definidas, o modelo da aplicação está completo. Esse modelo contempla os principais componentes da aplicação, a infra-estrutura de comunicação, os requisitos de QoS e políticas para dirigir adaptações. Todos esses aspectos de uma aplicação auto-adaptativa são descritos com a mesma linguagem de modelagem, criada a partir do meta-modelo da plataforma Meta-ORB. Para criar o modelo da aplicação, o desenvolvedor pode usar uma ferramenta de modelagem como

a do repositório de tipos, discutida no Capítulo 4. A Figura 5.11 mostra a definição dos modelos apresentados nas Figuras 5.7, 5.8, 5.9 e 5.10 como tipos no repositório através da ferramenta gráfica.

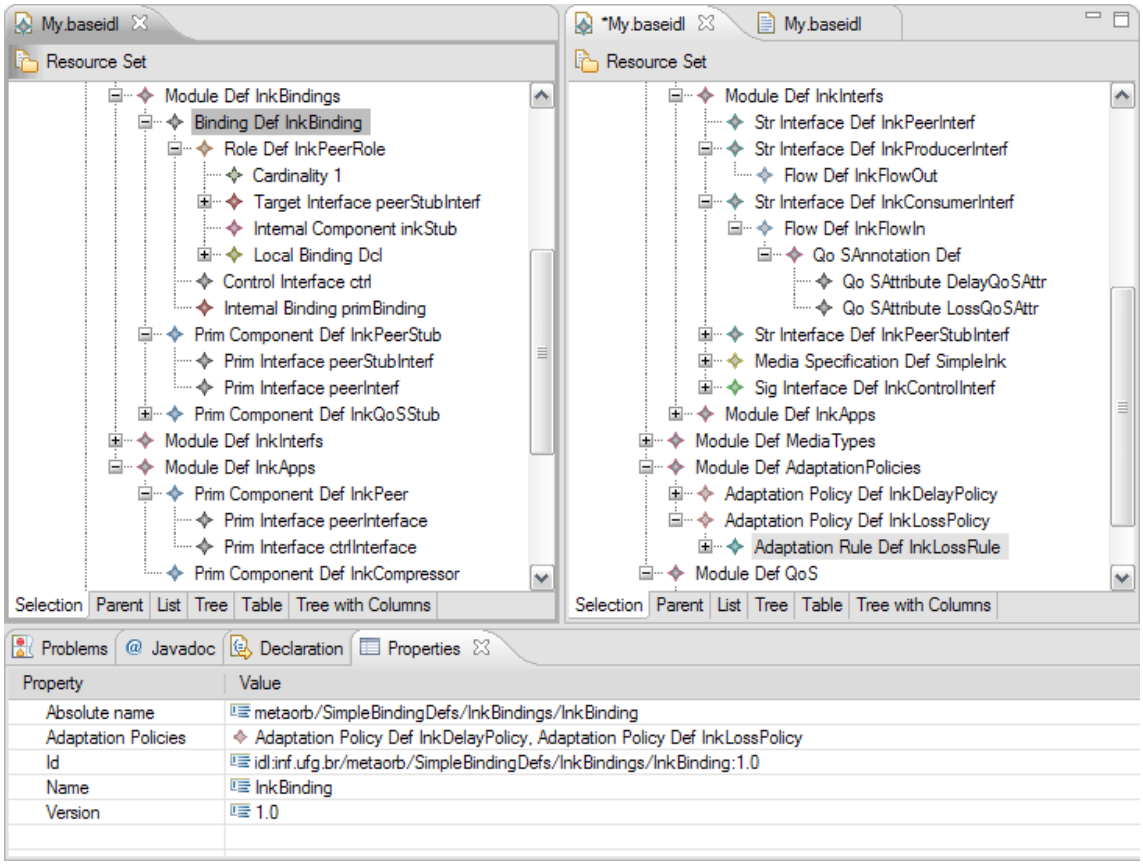



**Figura 5.11:** *Quadro Branco Compartilhado: Definição dos tipos.*

Após a definição dos tipos no repositório, o desenvolvedor precisa implementar os componentes e *bindings* primitivos que fazem parte da aplicação, caso eles não existam. O *binding PBMulticastUDP*, por exemplo, pode já existir e ser reutilizado, visto que sua implementação consiste em um mecanismo de comunicação primitivo. Assim, apenas os componentes *InkPeer* e *InkPeerStub* precisam ser implementados para instanciar a configuração inicial da aplicação. Entretanto, para possibilitar o comportamento auto-adaptativo, é preciso implementar os componentes que serão usados para adaptar a aplicação, como o componente *InkQoSStub*, e os interceptadores para monitorar os atributos de QoS, como *DelayInterceptor* e *LossInterceptor*.

A versão atual do protótipo MetaORB.NET, como visto no Capítulo 4, não conta com uma ferramenta para a geração automática de código a partir da definição de um tipo. Deste modo, ainda é preciso implementar as interfaces dos componentes, tal como mostra o Código 4.4. Em versões futuras, o código das interfaces e o esqueleto do código dos componentes poderão ser gerados automaticamente, diminuindo o esforço necessário para implementar as aplicações.

Além do código de implementação dos componentes e interfaces, existe ainda o código específico da aplicação, que engloba a implementação de uma interface gráfica apropriada e de funcionalidades para criar grupos colaborativos, configurar a tinta digital, etc. O Código 5.1 mostra alguns dos trechos do código da aplicação, que são explicados abaixo:

- Na linha 1, um objeto do tipo *InkOverlay* é criado. Esse objeto, da API *Microsoft.Ink*, fornece suporte para a coleta e renderização da tinta digital [65];
- Na linha 6, uma cápsula é inicializada, recebendo como nome o nome da máquina na qual está executando;
- Na linha 8, a fábrica de componentes é usada para criar localmente um componente do tipo *InkPeer*;
- Na linha 14, o serviço de nomes é usado para obter uma lista de referências de interfaces do tipo *InkPeerInterf*. Essa lista representa as interfaces dos componentes remotos que podem ser ligadas por um *binding* explícito, além de ser usada para selecionar os usuários que farão parte do grupo colaborativo;
- Na linha 20, a fábrica de *bindings* é usada para criar um *binding* do tipo *InkBinding* entre as interfaces selecionadas, formando assim um grupo colaborativo.

**Código 5.1** Criação de um binding do tipo InkBinding.

```
1. mainInkOver = new InkOverlay(Handle);
2. mainInkOver.Enabled = true;
3.
4. // ...
5.
6. capsule = new OrbCapsule(Environment.MachineName);
7.
8. capsule.CF.newC(
9.    "idl:inf.ufg.br/metaorb/SimpleBindingDefs/InkApps/InkPeer:1.0",
10.   "InkPeer", capsule.Uname, false, new object[] { mainInkOver, this });
11.
12. // ...
13.
14. var iRefList = capsule.NameService.lookupType(
15.    "idl:inf.ufg.br/metaorb/SimpleBindingDefs/InkInterfs/" +
16.    "InkPeerInterf:1.0");
17.
18. // ...
19.
20. var controlInterf = capsule.BF.newB(iRefList,
21.    "idl:inf.ufg.br/metaorb/SimpleBindingDefs/InkBindings/InkBinding:1.0",
22.    "InkBinding");
```

A Figura 5.12 mostra a interface gráfica do protótipo do quadro branco auto-adaptativo, desenvolvido com a plataforma MetaORB.NET. Esse protótipo é experimental e possibilita apenas a criação de grupos colaborativos e a comunicação da tinta digital em tempo real. Além disso, o middleware é capaz de se adaptar automaticamente de acordo com os requisitos de QoS da aplicação referentes ao atraso e a perda de pacotes.

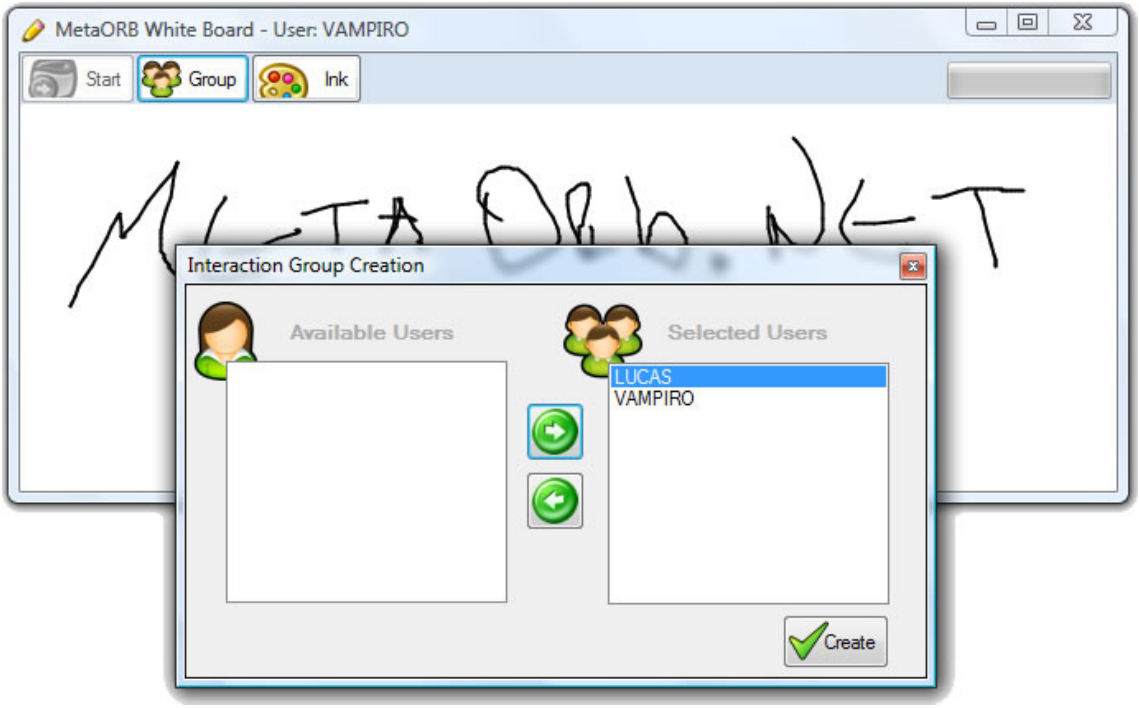



**Figura 5.12:** *Quadro Branco Compartilhado: protótipo auto-adaptativo.*

## 5.5 Avaliação

Na Seção 5.2.1 foram apresentados os problemas que podem afetar a interação em aplicações baseadas na comunicação da tinta digital em tempo real. Esses problemas foram observados no primeiro protótipo do quadro-branco e em experimentos preliminares envolvendo a comunicação da tinta digital em um ambiente móvel. O objetivo dos primeiros experimentos foi avaliar o impacto do atraso e da perda de pacotes nesse tipo de comunicação, que indicam a necessidade de auto-adaptação. Para avaliar o impacto das adaptações implementadas, uma nova série de experimentos da mesma natureza foram realizados. Como os anteriores, esses experimentos foram conduzidos no laboratório de tablet PCs do Instituto de Informática.

A mesma aplicação de monitoramento de fluxos de pacotes foi utilizada nos experimentos. Entretanto, essa aplicação foi modificada para possibilitar a recuperação de pacotes perdidos e a compressão dos pacotes enviados. O mecanismo de recuperação é baseado na numeração seqüencial dos pacotes. Assim, sempre que um pacote for perdido, o receptor solicita seu reenvio. O mecanismo de compressão utiliza a técnica discutida na Seção 5.3 para a compressão da tinta digital. Esses mecanismos são os mesmos empregados no quadro branco auto-adaptativo. Deste modo, é possível avaliar o fluxo de pacotes antes e após a realização da adaptação.

Para determinar de maneira mais precisa o efeito das adaptações, os experimentos foram realizados em cenários controlados e não em cenários reais como os experi-

mentos preliminares discutidos na Seção 5.2.1. Os experimentos foram conduzidos com diferentes números de nós conectados e em diferentes condições de tráfego de rede. A Tabela 5.1 mostra os dados coletados durante os experimentos em cada um dos cenários controlados. Uma análise desses dados é apresentada na Seção 5.5.1. Cada um dos experimentos consistiu no monitoramento de um fluxo de pacotes durante quatro minutos para cada um dos cenários. No cenário *A*, apenas dois tablets PCs estavam executando a aplicação de monitoramento. Nos cenários *B*, *C*, *D* e *E*, o número de nós conectados à rede aumentou para quatro, seis, oito e dez respectivamente.

| **Cenário** | **Data** | Exp 1 | Exp 2 | Exp 3 | Exp 4 |
|---|---|---|---|---|---|
| A | *Pacotes Perdidos* $(\%)$ | $0,32$ | $0,19$ | - | - |
| | *Atraso Médio* $(ms)$ | $8,25$ | $8,45$ | - | - |
| B | *Pacotes Perdidos* $(\%)$ | $0,71$ | $0,25$ | $1,68$ | - |
| | *Atraso Médio* $(ms)$ | $8,75$ | $9,03$ | $9,01$ | - |
| C | *Pacotes Perdidos* $(\%)$ | $2,07$ | $2,57$ | $5,41$ | $0,00$ |
| | *Atraso Médio* $(ms)$ | $85,85$ | $80,69$ | $50,79$ | $50,39$ |
| D | *Pacotes Perdidos* $(\%)$ | $2,48$ | $3,14$ | $3,75$ | $0,00$ |
| | *Atraso Médio* $(ms)$ | $88,93$ | $88,09$ | $84,23$ | $86,37$ |
| E | *Pacotes Perdidos* $(\%)$ | $4,08$ | $3$ | $4,48$ | $0,00$ |
| | *Atraso Médio* $(ms)$ | $89,69$ | $88,57$ | $87,27$ | $90,94$ |

**Tabela 5.1:** *Resultados experimentais para atraso e perdas de pacotes.*

Os experimentos *1* e *2* foram realizados no mesmo dia em horários diferentes, com a rede sendo utilizada apenas pelo aplicativo de monitoramento. O propósito desses experimentos é mostrar que, em cenários controlados e sob as mesmas condições, o comportamento de rede dificilmente muda, comprovando a estabilidade dos resultados obtidos nos experimentos *3* e *4*. Os experimentos *3* e *4* foram realizados no mesmo dia e em seqüência, e os nós conectados à rede estavam gerando tráfego à parte do tráfego gerado pelo aplicativo de monitoramento. O objetivo do experimento *3* é mostrar o efeito da carga da rede em comparação com os cenários base dos experimentos *1* e *2*. Já o experimento *4* mostra o efeito, após a adaptação, para comparação com os resultados do experimento *3*.

### 5.5.1 Resultados

No cenário *A*, a rede não estava sobrecarregada, resultando em um atraso médio e uma porcentagem de perdas pequenos. O gráfico da Figura 5.13 mostra os dados coletados neste cenário durante o experimento 2. Os pontos do gráfico representam os dados de atraso e perdas colhidos a cada intervalo de 10 pacotes. Neste cenário, a porcentagem total de perdas foi de $0,19\%$ e o atraso médio geral foi de $8,45ms$.

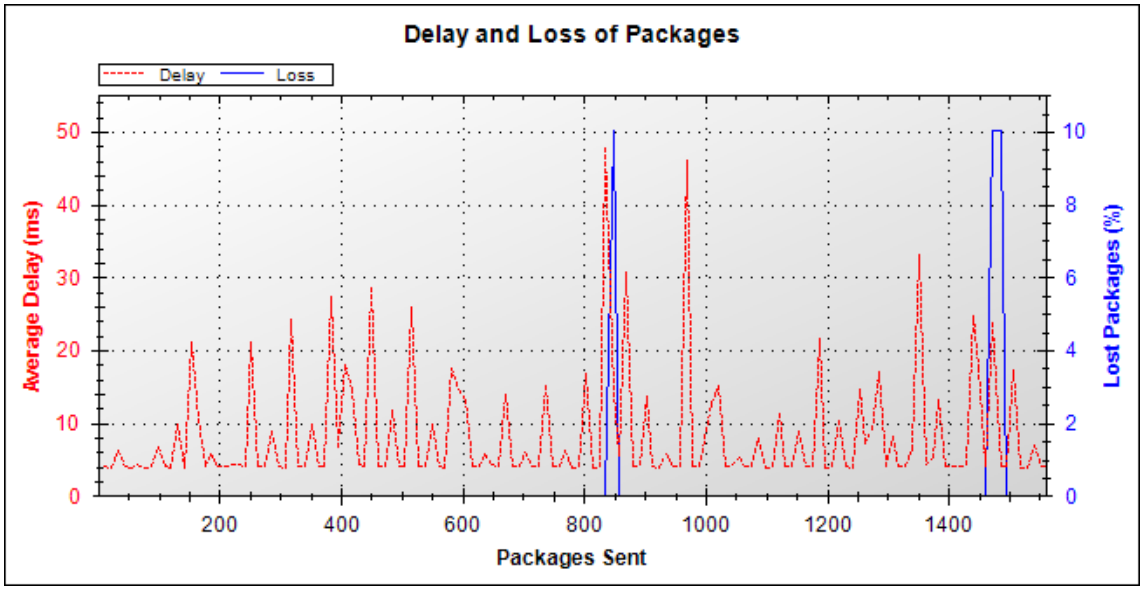



**Figura 5.13:** *Dados coletados: Cenário A, experimento 2*

Entretanto, no cenário E, durante o experimento 3, a porcentagem de perdas subiu para $4,48\%$ e o atraso médio chegou a $87.27ms$. Apesar do atraso ainda estar em um nível tolerável, essa porcentagem de perdas pode comprometer significativamente a interação entre os usuários da aplicação. O gráfico da Figura 5.14 mostra os dados coletados durante esse experimento.

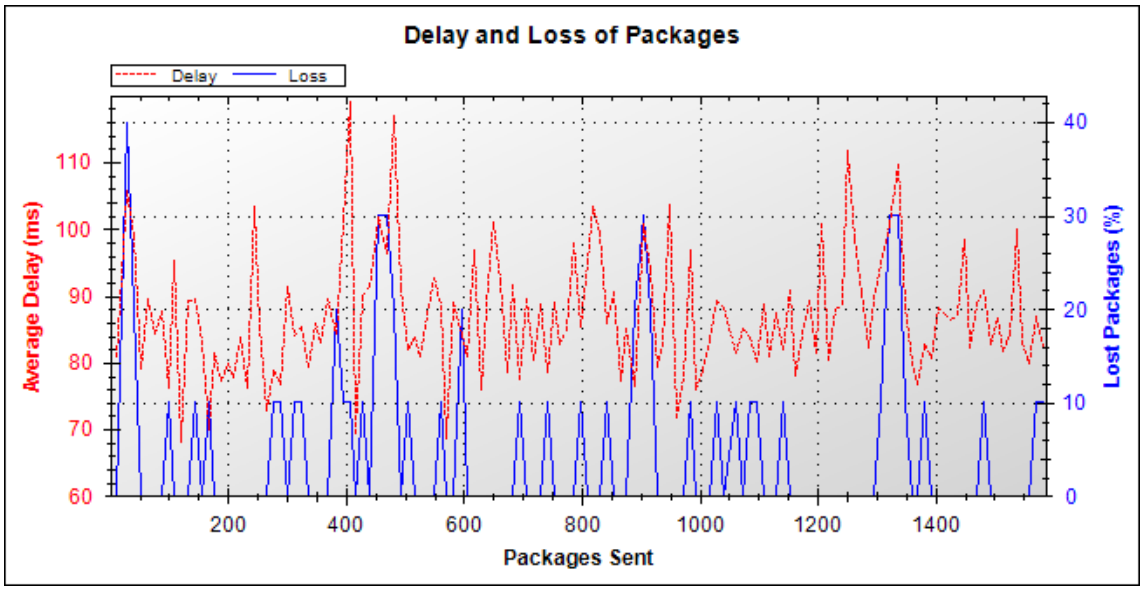



**Figura 5.14:** *Dados coletados: Cenário E, experimento 3*

Como visto na Seção 5.2.1, um simulador que permite visualizar em tempo real os efeitos do atraso e da perda de pacotes na comunicação da tinta digital foi desenvolvido. Os dados coletados no cenário *E* do experimento *3* foram usados para configurar esse simulador. Os efeitos do atraso e perda de pacotes nesse experimentos podem ser vistos

na Figura 5.15. Na figura, o painel superior é onde o usuário escreve com a tinta digital, e o painel inferior mostra o resultado da simulação.

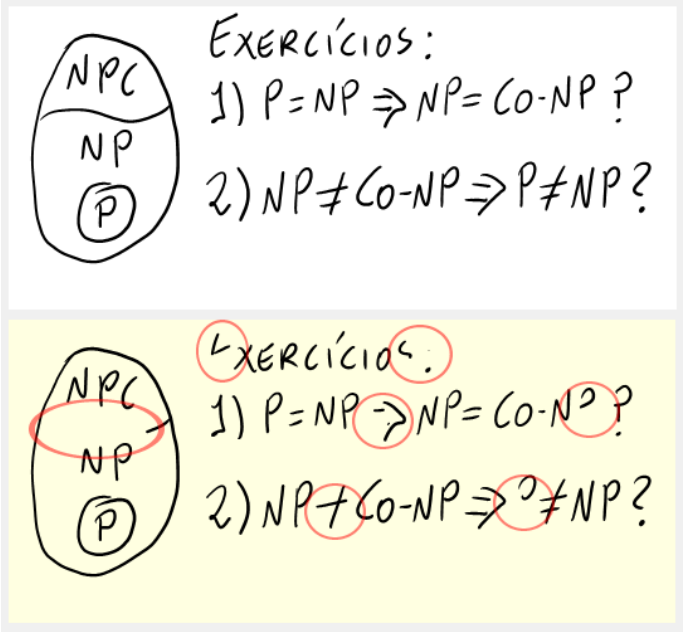



**Figura 5.15:** *Efeito da perda de pacotes de acordo com os dados do experimento 3, cenário E.*

O experimento *4* consistiu em introduzir a adaptação e repetir o experimento *3* em cada um dos cenários que apresentaram altas porcentagens de perdas. Apesar de apenas as perdas serem significativas, tanto o mecanismo de recuperação de pacotes perdidos quanto o de compressão de dados foram empregados. Com o mecanismo de recuperação, a porcentagem efetiva de perdas caiu para 0%, como esperado. O mecanismo de compressão, por sua vez, diminui o tamanho dos pacotes enviados pela rede, em uma tentativa de compensar o aumento do atraso médio causado pelo mecanismo de recuperação de perdas.

A Figura 5.16 mostra um gráfico com os dados coletados durante o experimento *4* no cenário *E*. No gráfico, valores negativos para a porcentagem de perdas representam a recuperação de pacotes perdidos, ou seja, uma porcentagem de $-20\%$ indica que, nos últimos dez pacotes recebidos, dois pacotes perdidos anteriormente foram recuperados. Neste experimento a porcentagem geral de perdas se manteve em 0% e o atraso médio foi de $90,94ms$, apenas $3,67ms$ a mais que o observado no experimento *3*, que não empregou adaptação. O experimento mostrou que, com a combinação dos dois mecanismos, não houve um aumento significativo no atraso e as perdas foram nulas.

Os experimentos mostrados na Tabela 5.1 foram conduzidos em cenários controlados, durante intervalos de tempo curtos e, por isso, os valores monitorados não apresentam mudanças subtas, como as observadas no experimento realizado em um cenário real (Figura 5.2). Em um cenário real, existe a necessidade do monitoramento constante dos

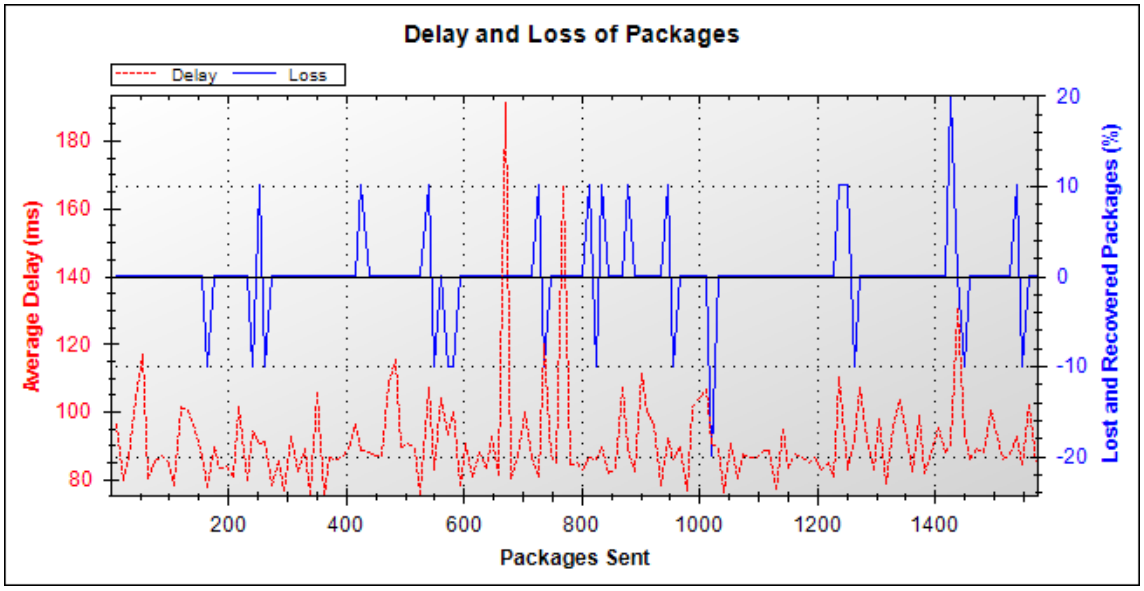



**Figura 5.16:** *Dados coletados: Cenario E, experimento 4 (Adaptação)*

níveis de atraso e de perdas para determinar quando a aplicação deve ser adaptada. Apesar disso, os benefícios demonstrados nos cenários controlados também são válidos para os cenários reais, onde a adaptação é realizada dinamicamente. Nos experimentos, apenas a adaptação em função de perdas de pacotes se mostrou necessária. Os efeitos da adaptação em um cenário onde a aplicação é prejudicada pelo atraso na recepção dos pacotes ainda precisam ser investigados.

## 5.6 Considerações Finais

Este capítulo apresentou um estudo de caso do emprego da arquitetura de auto-adaptação proposta, por meio do protótipo MetaORB.NET, no desenvolvimento de uma aplicação multimídia de tempo real com capacidades auto-adaptativas. A aplicação de estudo de caso foi desenvolvida pelo autor desta dissertação, que já está familiarizado com o modelo de programação da plataforma Meta-ORB, de forma que não foram identificadas maiores dificuldades na definição e implementação da aplicação. Assim, ainda é necessária uma avaliação do uso da arquitetura por outros desenvolvedores, menos familiarizados com o modelo de programação e com a tecnologia empregada.

Usando a arquitetura proposta, o desenvolvedor de aplicações auto-adaptativas pode seguir o método de desenvolvimento usado no estudo de caso apresentado neste capítulo, que estabelece os seguintes passos:

1. Estudo da aplicação e de seu ambiente - para definir os requisitos de qualidade de serviço e a necessidade de adaptação da aplicação;

2. Modelagem da aplicação - Definição ou reutilização dos tipos de componentes, interfaces, *bindings* e políticas que formam uma configuração particular da aplicação, o que pode ser feito com a ajuda da ferramenta de definição de tipos do repositório de tipos;
3. Implementação concreta - Implementação ou reutilização dos componentes e *bindings* primitivos que formam a aplicação e de interceptadores para monitoramento de QoS, caso necessário. Como visto na Seção 4.4 do Capítulo 4, essa etapa pode ser auxiliada por uma ferramenta de geração de código;
4. Implementação do código específico da aplicação - Isto pode envolver código para inicializar e instanciar o middleware juntamente com o código da interface gráfica da aplicação.

Este capítulo apresentou ainda, na Seção 5.5, uma avaliação experimental do impacto da adaptação na manutenção da qualidade de serviço em uma aplicação como o quadro branco. Os experimentos mostraram que, após a adaptação, as perdas foram nulas e o aumento no atraso não foi significativo. Entretanto, ainda são necessários experimentos para avaliar o custo introduzido pelo mecanismo de recuperação de erros e compressão de dados com respeito ao consumo extra de recursos (largura de banda, memória, processador, bateria). Além disso, ainda precisam ser avaliados os custos gerados pela infra-estrutura de gerenciamento de adaptação e monitoramento de QoS.

CAPÍTULO 6

# Trabalhos Relacionados

As principais contribuições deste trabalho se concentram em dois pontos. O primeiro ponto, relacionado à proposta de middleware reflexivo auto-adaptativo, é a definição de políticas de adaptação como parte do modelo da aplicação. O segundo, relacionado com o domínio da aplicação do estudo de caso, é o tratamento da tinta digital como um tipo de mídia em aplicações de multimídia distribuída.

Apesar de não ser o foco principal do trabalho, um estudo preliminar da tinta digital como um novo tipo de mídia foi apresentado no Capítulo 5, e constitui uma das motivações para a solução de auto-adaptação apresentada nesta dissertação. A tecnologia de tinta digital tem sido utilizada em diversas aplicações, como em [11, 1]. Várias delas são baseadas na comunicação da tinta digital através de uma rede, embora nenhuma delas trate adequadamente os aspectos de QoS envolvidos. Tendo em vista essa limitação, a tinta digital neste trabalho é tratada como um novo tipo de mídia, seguindo uma idéia já explorada em outros trabalhos [37, 58] e introduzindo requisitos de QoS nas aplicações baseadas nessa tecnologia.

As políticas são usadas como meta-informação para dirigir a adaptação do middleware e da aplicação. Neste trabalho, as políticas são especificadas de forma simplificada, pois foram projetadas para atender principalmente as aplicações multimídia com restrições de QoS, especialmente aquelas baseadas em tinta digital. Outros trabalhos, como [36] e [2], apresentam linguagens com sintaxe que permite a construção de políticas mais elaboradas. Um outro exemplo é a abordagem de *Quality Objects* [66], que define um conjunto de linguagens para a descrição de vários aspectos da aplicação, como especificações de QoS, monitoramento e adaptação.

Apesar da simplicidade das políticas de adaptação adotadas, o meta-modelo que as define é flexível e pode ser estendido para acomodar as necessidades de outros tipos de aplicação. Além disso, diferente das outras abordagens, o repositório de tipos pode ser usado no gerenciamento das políticas, possibilitando sua definição, armazenamento, distribuição e controle de versão, sem a necessidade de um serviço extra do middleware para isso. O restante deste capítulo discute trabalhos que apresentam abordagens para auto-adaptação de software alternativas àquela apresentada nesta dissertação.

## 6.1 Auto-adaptação Baseada em Modelos Arquiteturais

Grande parte dos trabalhos voltados à auto-adaptação de software defende em comum a separação do mecanismo de adaptação do restante da lógica da aplicação. Um exemplo é o *framework Rainbow* [27], cuja proposta é baseada em arquiteturas de software e em uma infra-estrutura reutilizável. Nesse *framework*, os modelos arquiteturais são usados em tempo de execução para expor algumas propriedades de interesse e aspectos comportamentais de um sistema.

O desenvolvedor pode adicionar comportamento auto-adaptativo a um sistema, definindo um estilo arquitetural para ele e especializando a infra-estrutura provida pelo *framework*. Um estilo inclui informações como os tipos de componentes e conectores que fazem parte do sistema, as propriedades de interesse para adaptação, restrições na composição dos tipos, operações e estratégias adaptativas, etc. Um modelo arquitetural é criado de acordo com o estilo arquitetural e com base em informações do sistema obtidas em tempo de execução. O *framework* pode então usar o modelo em tempo de execução para monitorar propriedades de interesse, avaliar violações de restrições e aplicar adaptações caso necessário.

Essa proposta se assemelha, em alguns pontos fundamentais, à proposta de auto-adaptação de middleware apresentada nesta dissertação e implementada no protótipo MetaORB.NET. O principal ponto é a utilização de modelos que representam o sistema adaptativo em tempo de execução. No *framework* Rainbow, esse modelo é criado como a combinação de um estilo arquitetural com informações do sistema em tempo de execução. Já na abordagem Meta-ORB, o modelo reflexivo é criado como uma combinação das meta-informações contidas nos tipos com informações do sistema em tempo de execução.

A diferença está na fonte estática das meta-informações, ou seja, os tipos e os estilos arquiteturais. Os tipos são baseados em um meta-modelo explícito que define o modelo de programação da plataforma, e são utilizados para a instanciação das entidades em tempo de execução que formam o sistema. Já os estilos são criados para representar as entidades do sistema, e não seguem um meta-modelo explícito ou têm relação direta com o processo de instanciação das entidades em tempo de execução. De maneira geral, o *framework* não define um modelo unificado para a criação estática e para a adaptação dinâmica do sistema. Isso se deve em grande parte a um de seus objetivos, que é a inclusão de comportamento auto-adaptativo para sistemas legados através da especialização de sua infra-estrutura.

A abordagem de auto-adaptação baseada na plataforma Meta-ORB não pode ser aplicada a sistemas legados que não sigam seu modelo de programação, a menos que estes sejam encapsulados em um componente. Mesmo restringindo sua aplicabilidade, a uniformidade dos modelos empregados na plataforma Meta-ORB pode facilitar sig-

nificativamente o processo de design e desenvolvimento de aplicações auto-adaptativas, como discutido no Capítulo 5, sem a necessidade de especializar a infra-estrutura do middleware. Já a abordagem do *framework* Rainbow pode ser aplicada a qualquer sistema, desde que a arquitetura do sistema possa ser descrita de acordo com um estilo arquitetural, o que inclui o conjunto de operações adaptativas para o qual o sistema provê suporte, e que o *framework* seja especializado.

Uma outra diferença está nas informações fornecidas pelo modelo em tempo de execução. No *framework* Rainbow, o modelo oferece informações sobre todas as propriedades definidas no estilo arquitetural do sistema, incluindo os valores para as propriedades de QoS que disparam a adaptação. Já na plataforma Meta-ORB, esse modelo, que é a auto-representação criada pelo mecanismo reflexivo, oferece apenas informações sobre a configuração estrutural do sistema, de acordo com o modelo de programação. Na arquitetura proposta, o monitoramento de QoS é realizado por um componente especializado que não faz parte do modelo reflexivo da plataforma e não o afeta. Desta forma, o monitor de QoS oferece um serviço do nível base para monitorar informações de contexto, que não refletem propriedades do sistema e sim do seu ambiente.

## 6.2 Auto-adaptação Baseada em Meta-tipos

*Chisel* [32] é um *framework* que permite a adaptação dinâmica dirigida por políticas em sistemas sensíveis ao contexto. O *framework* utiliza o conceito de meta-tipos oferecido por *Iguana/J*, uma extensão reflexiva para a linguagem Java [53]. A abordagem empregada é baseada na decomposição dos aspectos não-funcionais de um determinado tipo de objeto em múltiplos comportamentos possíveis. Esses comportamentos são implementados como meta-tipos de Iguana/J e podem ser associados aos objetos estatica ou dinamicamente. Um gerenciador de adaptações presente no meta-nível é responsável por adaptar a aplicação, selecionando e associando dinamicamente diferentes comportamentos aos objetos.

A proposta de Chisel é semelhante àquela apresentada nesta dissertação em alguns aspectos, tais como a utilização de mecanismos reflexivos para adaptação dinâmica e políticas para dirigir esses mecanismos. Porém, enquanto a proposta de Chisel é baseada em adaptações comportamentais, a do Meta-ORB é baseada em adaptações estruturais. A diferença entre as duas técnicas talvez esteja relacionada mais ao modelo de programação do que ao estilo de reflexão. Um meta-tipo associado a um objeto não altera a funcionalidade oferecida por um objeto, exposta através de seus métodos, e sim o comportamento de cada um desses métodos. Já os meta-componentes do Meta-ORB são capazes de alterar estruturalmente uma configuração de componentes. Da mesma forma, um componente pode ser substituído por outro que ofereça a mesma funcionalidade (que

expõe as mesmas interfaces) e possua comportamento diferente (implementação interna distinta).

O meta-nível de arquitetura na plataforma Meta-ORB permite a adaptação dinâmica tanto de componentes quanto de *bindings*, oferecendo acesso à estrutura interna dos diversos níveis de composição que formam essas entidades. Com isso, é possível reificar tanto um único componente quanto o *binding* que liga um sistema distribuído inteiro. Já os meta-tipos dos objetos não possuem as mesmas características de composição e distribuição, visando basicamente a reificação de objetos locais. Mesmo os eventos que disparam a adaptação são gerados localmente, com base principalmente em informações de contexto e recursos que afetam a aplicação localmente e não o sistema global.

As duas abordagens diferem ainda na caracterização do gerenciador de adaptação e dos monitores de contexto e recursos, que em Chisel fazem parte do meta-nível. Na arquitetura proposta nesta dissertação, o monitor de contexto e o gerenciador de adaptação são serviços do núcleo do middleware realizados por componentes do nível base. Esses componentes são dependentes do meta-nível para realizar suas tarefas mas não fazem parte do mesmo.

Assim como as políticas de adaptação apresentadas nesta dissertação, as políticas em Chisel são simplificadas para englobar apenas as informações necessárias ao tipo de adaptação para o qual o *framework* provê suporte. Chisel define uma linguagem declarativa própria para descrição de políticas, baseada em eventos, condições e ações. Já na arquitetura aqui proposta, o conceito de políticas de adaptação foi inserido no meta-modelo da plataforma. Apesar de também serem definidas de maneira declarativa, as políticas, assim como as outras construções do middleware, são definidas como tipos. Deste modo, a definição das políticas passa a ser parte do processo de modelagem do sistema, sem a necessidade da criação de uma nova linguagem para escrita de políticas na forma de *scripts*. Além disso, não é preciso um serviço extra do middleware para armazenar e recuperar políticas, o que é feito através do repositório de tipos.

## 6.3 Auto-adaptação Baseada em Reconfiguração Distribuída

Em [28], é apresentada uma meta-arquitetura que provê mecanismos para adaptação válida e segura de componentes que fazem parte de uma configuração distribuída de middleware. Essa arquitetura, assim como a apresentada nesta dissertação, é baseada na abordagem de middleware reflexivo de Open ORB. Deste modo, a solução para a adaptação dinâmica empregada consiste na reconfiguração de componentes através do mecanismo reflexivo oferecido pelo meta-espaço de arquitetura.

O trabalho defende que o modelo de reconfiguração local para *frameworks* de componentes oferecido por OpenCOM [20] também pode ser aplicado a topologias de componentes distribuídos. O conceito de *framework* de componente se assemelha ao de componente composto de Open ORB e Meta-ORB, ou seja, um componente que possui uma arquitetura interna formada por outros componentes. Cada *framework* de componente possui ainda uma infra-estrutura de suporte à realização de adaptações em sua arquitetura interna de forma válida e segura. Isso inclui uma interface de acesso ao meta-nível de arquitetura, além de mecanismos para verificar se o *framework* está em um estado seguro antes da adaptação e para validar a integridade da reconfiguração.

Para completar a infra-estrutura de auto-gerenciamento do *framework*, existem os configuradores, que são responsáveis por efetuar as adaptações. Cada *framework* está associado a um configurador, que mantém um conjunto de políticas locais e está conectado a uma máquina de contexto. Assim como no trabalho apresentado na Seção 6.2, as políticas utilizadas são *scripts* baseados em eventos, condições e ações.

Na proposta, a idéia de configuradores de *frameworks* locais foi estendida para configuradores associados a *frameworks* distribuídos. Um configurador distribuído está conectado a uma máquina de contexto global e possui um conjunto de políticas aplicáveis ao *framework* distribuído. As políticas definem reconfigurações que serão realizadas em cada um dos *frameworks* locais dos nós que formam o sistema.

A principal diferença entre a abordagem de auto-gerenciamento de middleware baseada em OpenCOM e aquela baseada em Meta-ORB é o alvo da adaptação. Enquanto o configurador age sobre um *framework* de componentes, o gerenciador de adaptação age sobre um *binding* explícito. O configurador pode ser aplicado tanto em *frameworks* locais quanto em distribuídos, já a abordagem baseada em Meta-ORB não define um mecanismo para auto-adaptação local de componentes compostos. Assim, somente configurações distribuídas podem ser gerenciadas. No modelo de programação de Meta-ORB, uma configuração de middleware distribuída é realizada por *binding* explícito e não existe o conceito de componente composto distribuído.

De maneira geral, a arquitetura proposta nesta dissertação está voltada para aplicações que envolvem a interação entre componentes remotos, sendo que essa interação é sensível a variações de QoS. Como os componentes interagem através de um *binding*, o alvo da reconfiguração é a arquitetura interna do *binding*. Deste modo, as aplicações que envolvem a reconfiguração de componentes compostos locais não são consideradas pois fogem ao escopo do trabalho. Apesar disso, em versões futuras, o meta-modelo pode ser estendido para suportar políticas de adaptação associadas a componentes compostos, e a arquitetura de auto-adaptação pode ser complementada com o gerenciamento local de adaptações.

Um outro aspecto importante que não foi tratado na arquitetura de auto-

adaptação para o Meta-ORB é a realização de adaptações de forma válida e segura. Uma validação estrutural da adaptação é feita pelo meta-componente antes de realizar a adaptação. Porém, não existe um mecanismo externo para checar e policiar as adaptações, como sugere o trabalho discutido nesta seção. Entretanto, o protocolo de auto-adaptação introduz um mecanismo para avaliar o efeito pós-adaptação. Esse mecanismo verifica se a adaptação surtiu o efeito desejado, o que implica na melhoria dos valores de QoS monitorados, e, caso contrário, desfaz a adaptação e retorna a configuração para seu estado anterior, descartando a política executada.

Existe ainda a checagem para determinar se o *framework* está em um estado seguro antes de realizar a adaptação, ou seja, nenhuma chamada está em execução no *framework* no momento da reconfiguração. Um mecanismo de bloqueio para leitura/escrita inserido em cada *framework* impede a adaptação de *frameworks* que estejam bloqueados. Já na plataforma Meta-ORB não existe um mecanismo semelhante e as adaptações são sempre executadas pelo meta-componente imediatamente, a menos que não sejam estruturalmente válidas. Para minimizar a perda de chamadas que passam por um *binding* que está sendo adaptado, as interfaces dos componentes que fazem parte do *binding* implementam um mecanismo de *buffer*. Assim, sempre que um *binding* local é desfeito, as chamadas que passam por esse *binding* são armazenadas no *buffer* da interface até que o *binding* seja refeito. Esse mecanismo não garante que todas as chamadas sejam efetivadas. Porém, em aplicações que envolvem fluxo contínuo de mídia, não é possível esperar o sistema entrar em um estado estacionário seguro antes de realizar a adaptação.

## 6.4 Auto-adaptação Baseada em Planejamento

A plataforma de middleware *Madam* [25], assim como os outros trabalhos discutidos neste capítulo, emprega modelos arquiteturais em tempo de execução em seu mecanismo de adaptação. O middleware é capaz de detectar variações de contexto que afetam a aplicação e realizar adaptações automaticamente. Porém, diferente das outras abordagens, o middleware não usa políticas de adaptação baseadas em eventos, condições e ações para dirigir as adaptações. No lugar das políticas são usadas *funções de utilidade*, que estabelecem apenas objetivos comportamentais, de modo que o próprio sistema determina quais ações devem ser tomadas para atingir esses objetivos.

Madam mantém dois modelos em tempo de execução. O primeiro consiste em um modelo da arquitetura do *framework*, oferecendo alternativas para componentes que podem ser encaixados no mesmo. O segundo é um modelo arquitetural da instância atual do *framework*, contendo os componentes atualmente em uso. Quando ocorre alguma mudança no contexto da aplicação, o gerenciador de adaptação do middleware incia um processo conhecido como *planejamento*, onde variações da arquitetura são derivadas e

avaliadas a partir do modelo da arquitetura do *framework*. As funções de utilidade são então aplicadas para cada uma das variações da arquitetura para determinar a de maior utilidade em função das propriedades da aplicação e de seu contexto. A variação de maior utilidade é então comparada com o modelo da instância atual para derivar os passos de reconfiguração, que são aplicados por um configurador.

Assim, o arquiteto do sistema não precisa definir, através de políticas, como o middleware deve proceder para realização automática de adaptações, embora precise definir as propriedades e as funções de utilidade de cada componente. As propriedades são definidas como anotações associadas aos tipos de componentes que formam um *framework*, assim como as anotações de QoS no Meta-ORB. As funções de utilidade estão associadas às possíveis implementações de um determinado tipo de componente. O benefício dessa abordagem é a possibilidade de derivar diversas variações para uma arquitetura e sempre será selecionada a de melhor utilidade de acordo com o contexto.

Já na abordagem proposta nesta dissertação, baseada em políticas de adaptação, existe apenas um número de variações equivalente ao número de políticas suportadas pelo *binding*, que são aplicadas de acordo com uma prioridade pré-definida. Essa abordagem não oferece garantias de que a adaptação realizada seja a melhor possível. Porém, o trabalho realizado pelo arquiteto do sistema é menos complexo e beneficiado pelo fato da linguagem e das ferramentas utilizadas na modelagem da aplicação e do middleware serem as mesmas utilizadas na definição das políticas de adaptação. Além disso, a derivação de variações do modelo arquitetural e sua avaliação através das funções de utilidade requerem um processamento maior do que o utilizado para avaliar as políticas de adaptação.

Nesta dissertação, a idéia defendida é a utilização da meta-informação contida nos tipos para dirigir adaptações dinamicamente. Para isto, o conceito de política de adaptação foi introduzido no meta-modelo da plataforma Meta-ORB. Por outro lado, as funções de utilidade oferecem uma alternativa interessante às políticas de adaptação, fazendo com que o próprio sistema determine as ações a serem realizadas. Uma possibilidade para trabalhos futuros é a introdução do conceito de função de utilidade no meta-modelo da plataforma Meta-ORB. Assim, o arquiteto do sistema pode escolher qual das duas abordagens se encaixa melhor em sua aplicação, ou mesmo trabalhar com as duas, utilizando, por exemplo, funções de utilidade para auto-adaptação de componentes compostos e políticas de adaptação para os *bindings*.

Um outro trabalho que utiliza adaptação baseada em planejamento é a plataforma *MUSIC* [55]. Um aspecto interessante desse trabalho é a auto-distribuição do processo de adaptação. Inicialmente, são empregadas funções de utilidade para determinar quais dispositivos do domínio são mais adequados para realizar cada um dos papéis envolvidos no processo de adaptação. Assim, os componentes que realizam os papéis de monitor de con-

texto, gerenciador de adaptação, configurador etc, são distribuídos automaticamente entre os diversos dispositivos do domínio da aplicação. A configuração distribuída que realiza o processo de adaptação pode ainda ser reconfigurada dinamicamente em decorrência de mudanças no contexto.

Na plataforma MetaORB.NET a requisição para monitoramento de QoS, a interpretação das políticas e a execução das reconfigurações são realizadas por um único componente presente na cápsula que criou o *binding*, que é o gerenciador de adaptação. Assim, o controle sobre o processo de adaptação é centralizado e fixo em um único dispositivo. Um problema desta abordagem é que uma falha na cápsula que controla o processo de auto-adaptação pode privar o sistema inteiro dessa funcionalidade. Estudos futuros envolvendo a arquitetura de auto-adaptação devem levar em consideração essa limitação, inserindo um mecanismo de tolerância a falhas, não só para o mecanismo de auto-adaptação mas também para os demais serviços fixos do middleware, como o serviço de nomes e o repositório de tipos.

CAPÍTULO 7

# Conclusões

Esta dissertação apresentou uma proposta para auto-adaptação baseada na plataforma de middleware reflexivo Meta-ORB. Em particular, essa abordagem emprega meta-modelagem para descrever as construções que são usadas na modelagem tanto das entidades que formam o middleware e as aplicações quanto de seu comportamento adaptativo. Os modelos resultantes são então usados em tempo de execução (como meta-informação) para instanciar o middleware e as aplicações e também para definir as ações realizadas pelo mecanismo de auto-adaptação.

Como artefatos concretos, o trabalho produziu um protótipo funcional do middleware auto-adaptativo (MetaORB.NET) e de serviços do middleware (repositório de tipos e serviço de nomes). Esses artefatos foram empregados em um estudo de caso que envolveu o desenvolvimento de uma aplicação auto-adaptativa. Essa aplicação é um quadro branco compartilhado baseado em tinta digital, capaz de se adaptar em função de variações do ambiente de rede que afetam a qualidade de serviço oferecida aos seus usuários.

As aplicações auto-adaptativas baseadas em tinta digital foram alvo de estudos desde o início deste trabalho, sendo tema de outras publicações do autor. Em [51], o potencial da arquitetura de auto-adaptação em sistemas colaborativos é investigado, em especial as aplicações colaborativas baseadas em tinta digital. Em [50], é apresentado um estudo preliminar da caracterização da tinta digital como um novo tipo de mídia. Já em [49], o emprego da meta-modelagem de políticas de adaptação em um middleware auto-adaptativo é discutido, usando novamente as aplicações de tinta digital como exemplo.

## 7.1 Principais Contribuições e Trabalhos Futuros

A seguir, é apresentada uma discussão sobre alguns dos resultados alcançados com este trabalho que, como visto no Capítulo 1, constituem algumas de suas principais contribuições. Além disso, são apresentadas possibilidades para trabalhos futuros derivados dessas contribuições.

### 7.1.1 Políticas de Adaptação

Empregando um middleware reflexivo, como o OpenORB ou o próprio Meta-ORB, o desenvolvedor precisa especificar o comportamento adaptativo de maneira procedural, como no Código 2.1 mostrado no Capítulo 2. Usando a abordagem defendida nesta dissertação, o comportamento adaptativo pode ser expresso através de políticas de adaptação especificadas declarativamente, como a mostrada na Figura 5.10. Com essa abordagem, as políticas tornam-se independentes do código da aplicação, facilitando sua especificação, modificação e reutilização.

As políticas de adaptação são descritas no meta-modelo da plataforma Meta-ORB, assim como as demais entidades usadas como meta-informação pelo middleware. Deste modo, a abordagem proposta emprega uma linguagem unificada para a modelagem do middleware e das aplicações, o que inclui os requisitos de QoS e as políticas de adaptação. Assim, o projeto de aplicações auto-adaptativas se torna mais fácil, uma vez que o desenvolvedor pode empregar a mesma linguagem e a mesma ferramenta para a modelagem da aplicação e de seu comportamento auto-adaptativo. Esse modelo, por sua vez, é usado para instanciar a aplicação e fornece a meta-informação usada pelo mecanismo auto-adaptativo.

As políticas de adaptação são definidas de maneira simplificada, sendo que ainda é necessário um estudo para tornar a sintaxe e semântica dessas políticas mais elaboradas, para atender outros tipos de aplicação, bem como as que envolvem outras formas de adaptação além da reconfiguração estrutural de um *binding*. Além disso, ainda é necessário investigar formas de compor as políticas. Atualmente, as políticas estão associadas aos *bindings*, de modo que a composição de *bindings* oferece uma maneira de compor hierarquicamente as políticas.

As políticas de adaptação constituem um exemplo de meta-informação que pode ser extraída de um modelo acessível em tempo de execução e usada pelo middleware como parâmetro de algum de seus serviços, neste caso o mecanismo de auto-adaptação. Trabalhos futuros podem pesquisar a modelagem de outros tipos de informações, como por exemplo as funções de utilidade discutidas no Capítulo 6.

### 7.1.2 Arquitetura de Auto-adaptação

A arquitetura da plataforma Meta-ORB foi estendida com um mecanismo para realizar as adaptações de forma dinâmica e automática, caracterizando a plataforma como um middleware reflexivo e auto-adaptativo. Um componente especializado no gerenciamento de adaptação foi criado para avaliar e aplicar as políticas de adaptação, trabalhando em conjunto com um componente especializado no monitoramento de QoS.

Esses componentes são definidos de forma abstrata na arquitetura do middleware, e suas implementações concretas são carregadas dinamicamente.

Na arquitetura atual, o gerenciador de adaptação é criado durante o processo de criação do *binding* e age de acordo com um protocolo bem estabelecido, discutido no Capítulo 3 para aplicar as políticas de adaptação. Trabalhos futuros podem envolver a definição de protocolos mais elaborados para avaliar e aplicar as políticas, pois elas podem se tornar mais elaboradas, envolvendo outros tipos de informações de contexto e outros tipos de adaptação.

O monitor de QoS utiliza estratégias baseadas na interceptação de mensagens para monitorar os valores de atraso e perdas de pacotes em uma aplicação. Estratégias diferentes ainda precisam ser desenvolvidas para atender aplicações que utilizam outros tipos de informações de contexto para se adaptar, como a mostrada no Código 2.1 que se adapta em função de mudanças na localização do usuário. Deste modo, trabalhos futuros podem investigar a integração da arquitetura de auto-adaptação com uma máquina de provisão de contexto mais completa.

### 7.1.3 Protótipo MetaORB.NET

Para demonstrar a viabilidade da arquitetura proposta, um protótipo do middleware foi desenvolvido. Esse protótipo, chamado MetaORB.NET, foi escrito com a linguagem C# na plataforma .NET, expandindo assim a aplicabilidade da plataforma, que contava anteriormente com protótipos em Java e Python. O protótipo foi desenvolvido apenas com as funcionalidades necessárias para demostrar a viabilidade da arquitetura. Entretanto, mesmo em sua versão atual, o protótipo pode ser empregado em cenários reais, como visto no estudo de caso apresentado no Capítulo 5.

O protótipo MetaORB.NET utiliza serviços web, implementados em Java, para acesso remoto ao repositório de tipos e ao serviço de nomes. Esses serviços facilitam a interoperabilidade entre protótipos escritos com diferentes linguagens, uma vez que oferecem acesso global ao repositório de tipos e ao serviço de nomes em um formato acessível por todos os protótipos. Este trabalho não envolveu a interação entre protótipos distintos, sendo que estudos de interoperabilidade ainda precisam ser realizados.

Apenas testes para verificar o funcionamento correto do protótipo foram realizados e sua usabilidade foi avaliada em um estudo de caso. O protótipo e seu mecanismo de auto-adaptação ainda precisam passar por uma avaliação de desempenho mais rigorosa. Essa avaliação pode determinar, por exemplo, o custo do gerenciamento de adaptação e do monitoramento de QoS em relação ao consumo de memória e banda, o tempo extra necessário para inicializar o mecanismo de auto-adaptação, o tempo gasto para realizar a adaptação, etc.

O protótipo MetaORB.NET é experimental, desenvolvido para avaliar e testar a arquitetura de auto-adaptação. Na Seção 4.4 do Capítulo 4 são discutidas algumas das funcionalidades que ainda precisam ser implementadas no protótipo. Assim como o protótipo, o repositório de tipos e a ferramenta de definição de tipos precisam ser aprimorados. Além disso, novas ferramentas podem ser projetadas para facilitar o desenvolvimento de aplicações auto-adaptativas, como o gerador automático de código, também discutido no Capítulo 4.

### 7.1.4 Tinta Digital como um Novo Tipo de Mídia

Um estudo preliminar sobre a caracterização da tinta digital como um novo tipo de mídia foi apresentado no Capítulo 5. Em aplicações distribuídas de tempo real, a tinta digital apresenta algumas características, como a sensibilidade a atraso e a perdas, semelhantes a outros tipos de mídia, como áudio e vídeo. Experimentos envolvendo a comunicação da tinta digital em ambientes móveis foram realizados e indicaram a necessidade de um tratamento adequado para manter a qualidade de serviço esperada pelos usuários de aplicações baseadas nessa tecnologia.

Trabalhos futuros ainda precisam ser realizados para determinar os níveis de QoS esperados por usuários de aplicações baseadas em tinta digital, para, por exemplo, identificar de forma mais precisa os limites toleráveis de atraso e de perdas para esse tipo de mídia em aplicações de tempo real. É necessário, ainda, investigar se outras informações de contexto têm relação com a qualidade de serviço em aplicações baseadas em tinta digital.

Inicialmente, para adaptar as aplicações baseadas em tinta digital, foram empregadas técnicas para recuperação de trechos perdidos e para compressão da tinta digital. Além dessas, outras formas de adaptação e seus efeitos na comunicação da tinta digital precisam ser estudadas. Por exemplo, parâmetros como o intervalo de envio de pacotes podem ser ajustados para modificar o comportamento do fluxo de mídia, ou o formato da tinta (resolução, cor, etc.) pode ser modificado para adequá-la a outros dispositivos.

### 7.1.5 Quadro Branco Compartilhado Auto-adaptativo

A tinta digital é uma tecnologia atualmente em ascensão, que pode ser empregada para aprimorar a interface de interação de diversos tipos de aplicação, como por exemplo as aplicações colaborativas. No Capítulo 5 foi apresentado um estudo de caso onde o protótipo MetaORB.NET (que implementa a arquitetura de auto-adaptação proposta) é utilizado no desenvolvimento de uma aplicação colaborativa baseada na tinta digital.

A aplicação consiste em um quadro branco compartilhado, onde um usuário pode fazer contribuições com a tinta digital que são transmitidas em tempo real para

os demais usuários. Essa aplicação, diferente de outras aplicações baseadas na tinta digital, leva em consideração os aspectos de QoS envolvidos na comunicação da tinta digital em um ambiente móvel e é capaz de se adaptar automaticamente em função de variações observadas nesse ambiente. O quadro branco, apesar de ser um protótipo contendo apenas as funcionalidades básicas para comunicação da tinta digital e criação de grupos colaborativos, pode ser usado em diversos cenários reais, como por exemplo em sala de aula para aprimorar a interação entre professor e aluno.

Trabalhos futuros podem envolver o desenvolvimento de melhorias para o quadro branco, como por exemplo a possibilidade de armazenar o estado atual do quadro antes de encerrar a aplicação e continuar a partir desse estado em uma execução futura ou a possibilidade de criar novos quadros e navegar entre eles (como em aplicações de apresentação de slides). Além disso, uma avaliação da usabilidade dessa aplicação ainda precisa ser feita em um cenário real, como uma sala de aula, para determinar, por exemplo, a satisfação dos usuários com a aplicação. O quadro branco é apenas um exemplo de aplicação, pesquisas futuras podem investigar a reutilização da mesma configuração de middleware usada nesta aplicação em outros tipos de aplicações colaborativas e até mesmo o desenvolvimento de aplicações que combinam o *binding* para transmissão da tinta digital com *bindings* para transmissão de outros tipos de mídia.

# Referências Bibliográficas


[1] ANDERSON, R.; ANDERSON, R.; SIMON, B.; WOLFMAN, S.; VANDEGRIFT, T.; YASUHARA, K. **Experiences with a Tablet PC Based Lecture Presentation System in Computer Science Courses**. *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, p. 56–60, 2004.

[2] ANTHONY, R. J. **Generic Support for Policy-Based Self-Adaptive Systems**. In: *DEXA '06: Proceedings of the 17th International Conference on Database and Expert Systems Applications*, p. 108–113, Washington, DC, USA, 2006. IEEE Computer Society.

[3] AREF, W.; BARBARA, D.; LOPRESTI, D.; TOMKINS, A. **Ink as a First-class Datatype in Multimedia Databases**. *Multimedia Databases. Springer-Verlag, New York*, 1995.

[4] AURRECOECHEA, C.; CAMPBELL, A.; HAUW, L. **A Survey of QoS Architectures**. *Multimedia Systems*, 6(3):138–151, 1998.

[5] BERQUE, D.; JOHNSON, D.; JOVANOVIC, L. **Teaching Theory of Computation Using Pen-based Computers and an Electronic Whiteboard**. *Proceedings of the 6th annual conference on Innovation and technology in computer science education*, p. 169–172, 2001.

[6] BÉZIVIN, J. **In Search of a Basic Principle for Model Driven Engineering**. *Novatica Journal, Special Issue, March-April*, 2004.

[7] BLAIR, G. S.; COULSON, G.; BLAIR, L.; DURAN-LIMON, H.; GRACE, P.; MOREIRA, R.; PARLAVANTZAS, N. **Reflection, Self-awareness and Self-healing in Open ORB**. In: *WOSS '02: Proceedings of the first workshop on Self-healing systems*, p. 9–14, New York, NY, USA, 2002. ACM.

[8] BLAIR, G.; COULSON, G.; ANDERSEN, A.; BLAIR, L.; CLARKE, M.; COSTA, F.; DURAN-LIMON, H.; FITZPATRICK, T.; JOHNSTON, L.; MOREIRA, R.; OTHERS. **The Design and Implementation of Open ORB 2**. *IEEE Distributed Systems Online*, 2(6):1–40, 2001.

[9] BLAIR, G.; COULSON, G.; ROBIN, P.; PAPATHOMAS, M. **An Architecture for Next Generation Middleware**. In: *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, p. 191–206. Springer-Verlag, 1998.

[10] BUDINSKY, F.; BRODSKY, S. A.; MERKS, E. **Eclipse Modeling Framework**. Pearson Education, 2003.

[11] CATTELAN, R.; TEIXEIRA, C.; RIBAS, H.; MUNSON, E.; PIMENTEL, M. **Inkteractors: Interacting with Digital Ink**. *Proceedings of the 2008 ACM symposium on Applied computing*, p. 1246–1251, 2008.

[12] CHEN, G.; KOTZ, D. **A Survey of Context-Aware Mobile Computing Research**. Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College, November 2000, 2000.

[13] CHIU, P.; KAPUSKAR, A.; REITMEIER, S.; WILCOX, L. **NoteLook: Taking Notes in Meetings with Digital Video and Ink**. *Proceedings of the seventh ACM international conference on Multimedia (Part 1)*, p. 149–158, 1999.

[14] CHRISTENSEN, E.; CURBERA, F.; MEREDITH, G.; WEERAWARANA, S. **Web Services Description Language (WSDL) 1.1**. "http://www.w3.org/TR/wsdl", acessado em Janeiro de 2009, 2001.

[15] COSTA, F. M. **Combining Meta-Information Management and Reflection in an Architecture for Configurable and Reconfigurable Middleware**. PhD thesis, University of Lancaster, 2001.

[16] COSTA, F. M.; SANTOS, B. S. **Structuring Reflective Middleware Using Meta-information Management: The Meta-ORB Approach and Prototypes**. *Journal of the Brazilian Computer Society (JBCS)*, 10:43—-58.

[17] COSTA, F. **Meta-ORB: A Highly Configurable and Adaptable Reflective Middleware Platform**. *Proceedings of the 20th Brazilian Symposium on Computer Networks*, p. 735–750, 2002.

[18] COSTA, F.; PROVENSI, L.; VAZ, F. **Using Runtime Models to Unify and Structure the Handling of Meta-information in Reflective Middleware**. *Lecture Notes in Computer Science*, 4364:232–241, 2007.

[19] COULSON, G. **What is reflective middleware**. *IEEE Distributed Systems Online*, 2(8):165–169, 2001.

[20] COULSON, G.; BLAIR, G.; GRACE, P.; JOOLIA, A.; LEE, K.; UEYAMA, J. **A Component Model for Building Systems Software**. In: *Proceedings of IASTED Software Engineering and Applications (SEA'04), Cambridge, MA, USA, Nov*, 2004.

[21] DEY, A.; ABOWD, G. **The Context Toolkit: Aiding the Development of Context-Aware Applications**. In: *Workshop on Software Engineering for Wearable and Pervasive Computing*, 2000.

[22] ECMA INTERNATIONAL. **Standard ECMA-334 - C# Language Specification**. "http://www.ecma-international.org/publications/standards/Ecma-334.htm", acessado em Dezembro de 2008, 2006.

[23] EFSTRATIOU, C.; CHEVERST, K.; DAVIES, N.; FRIDAY, A. **Architectural Requirements for the Effective Support of Adaptive Mobile Applications**. In: *Proceedings of the Second International Conference on Mobile Data Management*, 2001.

[24] FITZPATRICK, T.; BLAIR, G.; COULSON, G.; DAVIES, N.; ROBIN, P. **Supporting Adaptive Multimedia Applications through Open Bindings**. In: *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*, p. 128, Washington, DC, USA, 1998. IEEE Computer Society.

[25] FLOCH, J.; HALLSTEINSEN, S.; STAV, E.; ELIASSEN, F.; LUND, K.; GJØRVEN, E. **Using Architecture Models for Runtime Adaptability**. *IEEE SOFTWARE*, p. 62–70, 2006.

[26] FORMAN, G.; ZAHORJAN, J. **The Challenges of Mobile Computing**. *IEEE Computer*, 27(4):38–47, 1994.

[27] GARLAN, D.; CHENG, S.-W.; HUANG, A.-C.; SCHMERL, B.; STEENKISTE, P. **Rainbow: Architecture-Based Self-adaptation with Reusable Infrastructure**. *IEEE Computer*, 37(10):46–54, Oct. 2004.

[28] GRACE, P.; COULSON, G.; BLAIR, G.; PORTER, B. **A Distributed Architecture Meta-model for Self-managed Middleware**. *Proceedings of the 5th workshop on Adaptive and reflective middleware (ARM'06)*, 2006.

[29] HONG, J. I.; LANDAY, J. A. **SATIN: A Toolkit for Informal Ink-based Applications**. In: *UIST '00: Proceedings of the 13th annual ACM symposium on User interface software and technology*, p. 63–72, New York, NY, USA, 2000. ACM.

[30] HORN, P. **Autonomic Computing: IBM's Perspective on the State of Information Technology**. *IBM TJ Watson Labs, NY, 15th October*, 2001.

[31] HORSTMANN, M.; KIRTLAND, M. **DCOM Architecture**. "http://msdn.microsoft.com/en-us/library/ms809311.aspx", acessado em Novembro de 2008, 1997.

[32] KEENEY, J.; CAHILL, V. **Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Framework**. *IEEE International Workshop on Policies for Distributed Systems and Networks*, 0:3, 2003.

[33] KEPHART, J.; CHESS, D. **The Vision of Autonomic Computing**. *Computer*, 36(1):41–50, Jan 2003.

[34] KICZALES, G.; DES RIVIÈRES, J.; BOBROW, D. **The Art of the Metaobject Protocol**. MIT Press, 1991.

[35] KON, F.; ROMÁN, M.; LIU, P.; MAO, J.; YAMANE, T.; MAGALHÃES, C.; CAMPBELL, R. H. **Monitoring, Security, and Dynamic Configuration with the DynamicTAO Reflective ORB**. In: *Middleware '00: IFIP/ACM International Conference on Distributed systems platforms*, p. 121–143, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[36] LOBO, J.; BHATIA, R.; NAQVI, S. **A Policy Description Language**. *Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence table of contents*, p. 291–298, 1999.

[37] LOPRESTI, D. **Ink as Multimedia Data**. *Proceedings of the Fourth Intl. Conference on Information, Systems, Analysis and Synthesis, July*, p. 122–128, 1998.

[38] MAES, P. **Concepts and Experiments in Computational Reflection**. *ACM SIGPLAN Notices*, 22(12):147–155, 1987.

[39] MICROSOFT CORPORATION. **Microsoft .NET Framework**. "http://www.microsoft.com/net/", acessado em Dezembro de 2008, 2008.

[40] MILI, H.; PACHET, F.; BENYAHIA, I.; EDDY, F. **Metamodeling in OO: OOPSLA'95 workshop summary**. *SIGPLAN OOPS Mess.*, 6(4):105–110, 1995.

[41] OBJECT MANAGEMENT GROUP. **CORBA Component Model Specification v4.0**. "http://www.omg.org/technology/documents/formal/components.htm", acessado em Novembro de 2008, 2006.

[42] OBJECT MANAGEMENT GROUP. **Meta Object Facility (MOF) Core Specification**. "http://www.omg.org/spec/MOF/2.0/", acessado em Novembro de 2008, 2006.

[43] OBJECT MANAGEMENT GROUP. **OMG Unified Modeling Language (OMG UML), Infrastructure**. "http://www.omg.org/spec/UML/2.1.2/", acessado em Novembro de 2008, 2007.

[44] OBJECT MANAGEMENT GROUP. **XML Metadata Interchange (XMI)**. "http://www.omg.org/spec/XMI/2.1.1/", acessado em Novembro de 2008, 2007.

[45] OKAMURA, H.; ISHIKAWA, Y.; TOKORO, M. **AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework**. In: *Proceedings of the Workshop on New Models for Software Architecture*, 1992.

[46] PARASHAR, M.; HARIRI, S. **Autonomic Computing: An Overview**. *LECTURE NOTES IN COMPUTER SCIENCE*, 3566:257, 2005.

[47] POON, A.; WEBER, K.; CASS, T. **Scribbler: A Tool for Searching Digital Ink**. *Conference on Human Factors in Computing Systems*, p. 252–253, 1995.

[48] PROVENSI, L. L. **Uma Infra-Estrutura de Suporte a Binding Explícito na Plataforma MetaORB4Java**. Trabalho de conclusão de curso, Universidade Federal de Goiás, 2006.

[49] PROVENSI, L. L.; COSTA, F. M.; SACRAMENTO, V. **Self-adaptive Middleware for Digital Ink Based Applications**. In: *ARM '08: Proceedings of the 7th workshop on Reflective and adaptive middleware*, p. 29–34, New York, NY, USA, 2008. ACM.

[50] PROVENSI, L. L.; COSTA, F. M.; SACRAMENTO, V. **Tinta Digital em Aplicações Multimídia para Ambientes Móveis**. In: *WebMedia '08: Proceedings of the XIV Simpósio Brasileiro de Sistemas Multimídia e Web (Short Paper)*, p. 49–52. SBC, 2008.

[51] PROVENSI, L. L.; COSTA, F. M.; SACRAMENTO, V. **Uma Arquitetura de Middleware para Suporte a Aplicações Colaborativas de Tinta Digital**. In: *SBSC '08: Proceedings of the 2008 Simpósio Brasileiro de Sistemas Colaborativos*, p. 180–191, Washington, DC, USA, 2008. IEEE Computer Society.

[52] PYTHON SOFTWARE FOUNDATION. **Python Programming Language**. "http://www.python.org/", acessado em Novembro de 2008, 2008.

[53] REDMOND, B.; CAHILL, V. **Iguana/J: Towards a Dynamic and Efficient Reflective Architecture for Java**. In: *ECOOP 2000 Workshop on Reflection and Metalevel Architectures, June*, 2000.

[54] RIEFFEL, E.; TOOMEY, L. **Systems and Methods for Generating and Controlling Temporary Digital Ink**, Oct. 23 2007. US Patent 7,286,141.

[55] ROUVOY, R.; BEAUVOIS, M.; LOZANO, L.; LORENZO, J.; ELIASSEN, F. **MUSIC: An Autonomous Platform Supporting Self-adaptive Mobile Applications**. In: *MobMid '08: Proceedings of the 1st workshop on Mobile middleware*, p. 1–6, New York, NY, USA, 2008. ACM.

[56] SACRAMENTO, V.; ENDLER, M.; RUBINSZTEJN, H.; LIMA, L.; GONCALVES, K.; NASCIMENTO, F.; BUENO, G. **MoCA: A Middleware for Developing Collaborative Applications for Mobile Users**. *Distributed Systems Online, IEEE*, 5(10):2–2, 2004.

[57] SCHMIDT, D. **Guest Editor's Introduction: Model-driven Engineering**. *Computer*, 39(2):25–31, 2006.

[58] SENI, G.; YAEGER, L.; TREMBLAY, C.; FRANKE, K.; OTHERS. **Ink Markup Language (InkML)**. "http://www.w3.org/TR/InkML/", acessado em Novembro de 2008, 2006.

[59] SUN MICROSYSTEMS. **Java 2 Platform, Micro Edition**. "http://java.sun.com/javame/index.jsp", acessado em Novembro de 2008, 2008.

[60] SUN MICROSYSTEMS. **Java Remote Method Invocation**. "http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp", acessado em Novembro de 2008, 2008.

[61] SZYPERSKI, C.; GRUNTZ, D.; MURER, S. **Component Software - Beyond Object-Oriented Programming**. Addison-Wesley and ACM, 2nd edition, 2002.

[62] THE ECLIPSE FOUNDATION. **Eclipse Modeling Framework Project (EMF)**. "http://www.eclipse.org/modeling/emf/", acessado em Janeiro de 2009, 2009.

[63] VAN GIGCH, J. **System Design Modeling and Metamodeling**. Plenum Pub Corp, 1991.

[64] VOGEL, A.; KERHERVE, B.; VON BOCHMANN, G.; GECSEI, J. **Distributed Multimedia and QOS: A Survey**. *Multimedia, IEEE*, 2(2):10–19, 1995.

[65] YAO, P. **Tablet PC: Add Support for Digital Ink to Your Windows Application**. *MSDN Magazine. The Microsoft Journal for Developers. http://msdn. microsoft. com/msdnmag*, 2004.

[66] ZINKY, J.; BAKKEN, D.; SCHANTZ, R. **Architectural Support for Quality of Service for CORBA Objects**. *THEORY AND PRACTICE OF OBJECT SYSTEMS*, 3(1):55–73, 1997.