Adilson Barboza Lopes

Um *Framework* para Configuração e Gerenciamento de Recursos e Componentes em Sistemas Multimídia Distribuídos Abertos

Tese de Doutorado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos para obtenção do título de Doutor em Engenharia Elétrica. Área de concentração: Engenharia de Computação.

Orientador: Maurício Ferreira Magalhães

Co-orientador: Glêdson Elias da Silveira

Campinas, SP
2006

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA E ARQUITETURA - BAE - UNICAMP

Lopes, Adilson Barboza
L881u Um framework para configuração e gerenciamento de recursos e componentes em sistemas multimídia distribuídos abertos / Adilson Barboza Lopes. --Campinas, SP: [s.n.], 2006.

Orientador: Maurício Ferreira Magalhães.
Tese (doutorado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Sistemas multimídia. 2. Middleware. 3. Componentes de software. 4. Sistemas operacionais distribuídos (Computadores). 5. Multimídia interativa. 6. Televisão interativa. I. Magalhães, Maurício Ferreira. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Titulo em Inglês: A framework for configuration and management of resources and components in open distributed multimedia systems.
Palavras-chave em Inglês: Software components, Multimedia distributed systems, Adaptive middleware, Interactive digital television.
Área de concentração: Engenharia de Computação.
Titulação: Doutor em Engenharia Elétrica
Banca examinadora: Eleri Cardozo, Ivan Luiz Marques Ricarte, Nelson Souto Rosa, Noemi de La Rocque Rodriguez.
Data da defesa: 30/08/2006

Adilson Barboza Lopes

Um *Framework* para Configuração e Gerenciamento de Recursos e Componentes em Sistemas Multimídia Distribuídos Abertos

Tese de Doutorado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos para obtenção do título de Doutor em Engenharia Elétrica. Área de concentração: Engenharia de Computação.

Banca Examinadora:
Prof. Dr.Eleri Cardozo - UNICAMP
Prof. Dr. Ivan Luiz Marques Ricarte – UNICAMP
Prof. Dr. Maurício Ferreira Magalhães - UNICAMP
Prof. Dr. Nelson Souto Rosa - UFPE
Profa. Dra. Noemi Rodriguez – PUC-Rio

Campinas, SP
2006

Resumo

Em sistemas multimídia distribuídos existe uma diversidade de dispositivos de hardware, sistemas operacionais e tecnologias de comunicação. Para tratar os requisitos destas aplicações, os componentes do sistema precisam interagir entre eles considerando os aspectos de QoS de cada um dos elementos envolvidos. Neste contexto, esta tese apresenta o Cosmos – um *framework* baseado em componentes proposto para dar suporte à configuração e gerenciamento de recursos em sistemas multimídia. Como prova de conceito do Cosmos, o *framework* definido foi usado no projeto do *middleware* AdapTV – um *middleware* para sistemas de televisão digital interativa. O projeto do AdapTV explora os principais componentes dos modelos que foram definidos no Cosmos: o modelo de descrição de aplicações de forma independente de linguagens; o modelo de interconexão, que trata as questões de comunicação entre componentes heterogêneos usando diferentes tecnologias de comunicação; e o modelo de gerenciamento de QoS, que permite o monitoramento e a adaptação do sistema. Estes modelos foram explorados na implementação de um protótipo do *middleware* AdapTV e de uma aplicação distribuída que realiza a captura, transmissão e apresentação de um fluxo de vídeo. Para dar suporte à reusabilidade, o modelo explora o conceito de propriedades para estabelecer acordos de configuração (estáticos e dinâmicos) envolvendo negociações entre os requisitos dos componentes e as características da plataforma.

Palavras-chave: Componentes de *software*, sistemas multimídia distribuídos, *middleware* adaptativo, televisão digital interativa.

Abstract

Distributed multimedia applications involve a diversity of hardware devices, operating systems and communication technologies. In order to fulfill the requirements of such applications, their constituting components need to interact with each other, as well as to consider QoS issues related to devices and transmission media. In such a context, this thesis presents the Cosmos component-based framework for configuration and management of resources of open, distributed multimedia systems. As a proof of concept, the framework was used in the design of the AdapTV middleware – a middleware for interactive television which explores the major components of the Cosmos, including: the model to describe and represent applications independently of language aspects; the interconnection model that allows communication between components in heterogeneous and distributed multimedia environments; and the QoS management model that provides support for adaptation in the middleware layer, triggered by QoS and user requirements changes. These models have been explored in the implementation of a prototype, which includes the AdapTV middleware and a distributed application example that captures, transmits and presents a video flow. In order to provide a generic and reusable approach, and to establish configuration agreements among component requirements and platform features, the framework explores the concept of properties.

Keywords: Software components, multimedia distribute systems, adaptive middleware, interactive digital television.

Quem não pode fazer grandes coisas,
faça ao menos o que estiver na medida de suas forças;
certamente não ficará sem recompensa.

Santo Antônio de Pádua

<u>Para tudo há momento e tempo</u>:

tempo de dar a luz
e tempo de morrer;
tempo de plantar
e tempo para arrancar o
que se plantou;
tempo de matar
e tempo de curar;
tempo de demolir
e tempo de construir;
tempo de chorar
e tempo de rir;
tempo de lamentar
e tempo de dançar;
tempo de atirar pedras
e tempo de juntar pedras;
tempo de abraçar
e tempo de evitar o abraço;
tempo de procurar
e tempo de perder;
tempo de guardar
e tempo de jogar fora;
tempo de rasgar
e tempo de costurar;
tempo de calar
e tempo de falar;
tempo de amar
e tempo de odiar;
tempo de guerra
e tempo de paz;

Ele faz tudo belo a seu tempo,
e dá ao coração humano até o sentido do tempo,
sem que o homem possa descobrir a obra que Deus fez do começo ao fim.

Eclesiastes (3, 1-8; 11-12)

Agradecimentos

A Deus, pelo dom da vida. É graças a Ele que encontramos a sabedoria, o conhecimento e a ciência da lei: realizar boas ações. “A ciência do sábio espalha-se como a água que transborda, e o conselho que ele dá permanece como fonte de vida” (Eclesiástico, 21, 16).

À Iracy, minha esposa, e aos meus filhos, Marina e Maurício, pelo incentivo e paciência. Obrigado por compreenderem a minha ausência em tantos momentos importantes de suas vidas.

Aos meus familiares e amigos, sobretudo aos meu pais, pelos valores que me passaram. Aos meus queridos irmãos, muito obrigado pelo incentivo e orações que tanto me deram forças para superar os momentos de desânimo e dificuldades.

Aos meus orientadores, Professores Mauricio e Gledson, pelas orientações que me permitiram realizar este trabalho. Desde a orientação da minha dissertação de mestrado o Prof. Maurício tem sido mais que um orientador. Ele tem proporcionado uma convivência pautada por uma conduta profissional exemplar e, acima de tudo, uma grande amizade. Esta conduta foi fundamental para que o desânimo de alguns momentos não me deixasse desistir, nem tampouco afetar a nossa amizade pessoal. Com certeza esta convivência me fez amadurecer nos aspectos pessoal e profissional. Quanto ao Prof. Gledson gostaria de destacar o quanto aprendi com ele nestes últimos anos. A sua postura profissional e sua vibração constante me contagiaram a ponto de descobrir potenciais que não imaginava tê-los. O Prof. Gledson também me ensinou muito com a sua visão e autenticidade na forma de conduzir suas ações, sobretudo com ética e muita garra. Além da orientação, obrigado pela amizade, Prof. Gledson.

Aos amigos e colaboradores Frederico Borelli e Carlos Eduardo da Silva (conhecido por todos como “Carioca”), por terem acreditado nas idéias propostas e terem investido esforços de seus trabalhos de mestrado no contexto do meu projeto de doutoramento. Agradeço também aos demais alunos e colaboradores que estiveram envolvidos neste projeto, Frederico Amaro, Frederico Lopes, Diogo Oliveira e José Augusto. A colaboração de vocês na implementação do protótipo foi muito importante.

Aos membros da banca examinadora, Professores Eleri, Ivan, Nelson e Noemi, por terem aceito o convite e pelas valiosas sugestões.

Ao Departamento de Informática e Matemática Aplicada, pelo apoio através de suas chefias, desde a época do prof. Márcio Muniz até o atual chefe, Prof. Ivan.

Aos colegas e amigos do DIMAp, pelo incentivo e paciência e por terem assumido o peso de meus afastamentos com o aumento de suas respectivas cargas horárias.

Não poderia, sob pena de ser injusto, deixar de fazer um agradecimento especial a dois professores e amigos: Pedro Maia e Guido Lemos.

O Pedro e a Graça, sua esposa, mesmo sem me conhecerem, me "adotaram" em Natal; na verdade, não adotaram só a mim, mas a toda a minha família. Quantas vezes eles chegaram a disponibilizar recursos particulares para que pudesse me deslocar até a UNICAMP. Este apoio material, e acima de tudo este acolhimento foram fundamentais. Obrigado por tudo, Pedro e Graça.

O Guido sempre esteve presente no desenvolvimento deste trabalho com a sua amizade e o seu incentivo. A sua confiança, demonstrada através do apoio dos projetos HiTV e FlexTV, foram fundamentais para a realização e desenvolvimento do protótipo. Por isso, Guido, o meu agradecimento especial.

Obrigado aos funcionários do DIMAp, pela amizade e pelo apoio diante das minhas necessidades. Em especial, o meu agradecimento à Graça, não somente pela amizade, mas também pelas "cobranças".

Para finalizar, meus agradecimentos à FEEC/UNICAMP e à CAPES, pela oportunidade que me deram para desenvolver esta tese.

À Iracy, minha esposa,

e à Elvira, minha mãe que, mesmo ausente, sei o quanto está feliz por esta realização.

Sumário

Lista de Figuras

Glossário

ADL:	Architectural Description Language
API :	Application Programming Interface
CCM:	CORBA Component Model
CM-tel :	Component Model for Telematic Applications
CODEC:	Codificador/Decodificador – dispositivo de hardware ou software que codifica/decodifica sinais
CORBA:	Common Object Request Broker Architecture
COTS:	Commercial-off-the-shelf (COTS)
DCOM:	Distributed Component Object Model
EJB:	Enterprise Java Beans
IDL:	Interface Definition Language
ISO:	International Organization for Standardization
ITU-T:	International Telecommunication Union –Telecommuniction Standardization Sector
J2EE:	Java 2 Enterprise Edition
J2SE:	Java 2 Standard Edition
JMF:	Java Media Framework
LAN:	Local Area Network
MDA:	Model Driven Architeture
OMG:	Object Management Group
ORB:	Object Request Broker
PDA:	Personal Digital Assistant – Assistente Pessoal Digital (computador de dimensões reduzidas)

QoS:	Quality of Service
RM-ODP:	Reference Model for Open Distributed Processing
RMI:	Remote Method Invocation
RPC:	Remote Procedure Call
RT-CORBA:	Real Time CORBA
TCP/IP :	Transfer Control Protocol/Internet Protocol
TINA-C:	Telecommunication Information Network Architecture Consortium
UDP:	User Datagram Protocol
UML:	Unified Modeling Language
W3C :	World Wide Web Consortium
WWW:	World Wide Web

Capítulo 1

Introdução

Este capítulo apresenta uma visão geral da presente tese onde são abordados vários aspectos relacionados às tecnologias atuais para desenvolvimento e suporte à execução de sistemas multimídia distribuídos. O trabalho analisa e propõe soluções adaptáveis e configuráveis baseadas nos conceitos de componentes de *software* e computação reflexiva no escopo da camada de *middleware*. O objetivo é gerar uma proposta inovadora que atenda os requisitos de novas aplicações, como por exemplo, sistemas de televisão digital interativa.

1.1 Sistemas Multimídia Distribuídos

A integração das telecomunicações com a computação e a multimídia transformou o computador em um veículo de comunicação. Neste panorama de convergência digital, que tem ocorrido principalmente devido à popularidade da Internet, o caráter distribuído das aplicações foi fortemente impulsionado, permitindo o desenvolvimento de soluções nos mais variados domínios, tais como: televisão interativa, tele-medicina, educação à distância e videoconferência.

Uma aplicação multimídia pode manipular diversas mídias com requisitos temporais críticos, tais como áudio e vídeo [1]. A complexidade destas aplicações é incrementada, uma vez que poderão ser executadas em dispositivos com capacidades variadas de processamento, armazenamento, qualidade de serviço (QoS - *Quality of Service*) e diferentes interfaces com o usuário. Em particular, o desafio de explorar a tecnologia multimídia no contexto da *Internet* se torna ainda maior, pois há uma enorme diversidade de requisitos associados e uma variação imprevisível de QoS devido às características de conectividade à rede. Por outro lado, as aplicações possuem diferentes estilos de interfaces de interação tornando a implementação dependente dos tipos de dispositivos e das tecnologias de acesso empregadas.

Muito esforço tem sido empreendido para prover facilidades, serviços e plataformas de suporte para estes sistemas. O conceito de *middleware* [2] está presente, tanto em abordagens para suportar o desenvolvimento de sistemas distribuídos [3] [4] [5] [6], como para sistemas multimídia [1] [7] [8]. Entretanto, as características inerentes destes sistemas são extremamente variáveis, implicando em novos requisitos à medida que novas tecnologias de *hardware* e *software* são disponibilizadas. Dessa forma, é importante criar condições para que o *middleware* realize, automaticamente, o ajuste e a adaptação entre as interfaces de programação disponibilizadas pelo *middleware* e as possíveis plataformas de *hardware* e sistemas operacionais, de maneira a não comprometer a portabilidade e a reusabilidade da aplicação.

Vários trabalhos têm discutido a questão da adaptação focada em aspectos relacionados com a variação da QoS [7] [8] [9]. A maioria dessas soluções foram definidas na camada de *middleware* através do paradigma de objetos [10] e baseadas em tecnologias como CORBA [4], DCOM [5] e Java-RMI [6]. Entretanto, estas soluções ainda incorporam a visão de um sistema centralizado onde o processamento se concentra no servidor e a rede tem o seu papel limitado a transmitir dados [11] [12].

Paralelamente ao esforço da indústria na busca de novas soluções, a ISO (*International Organization for Standardization*) e a ITU-T *(International Telecomunication Union-Telecomunication Standardization Sector*) definiram um padrão para uniformizar o processo de desenvolvimento de sistemas distribuídos. Como resultado deste esforço, foi definido o Modelo de Referência para Processamento Distribuído Aberto (RM-ODP) [13]. Este modelo pode ser considerado um metamodelo para processamento distribuído aberto, cujos conceitos podem ser aplicados em diferentes domínios, fornecendo assim a base conceitual para a realização de diferentes instanciações. Várias soluções atualmente apresentadas na literatura exploram os conceitos definidos nesse modelo. Entre outras, podemos citar: TINA-C (*Telecomunication Information Network Architecture Consortium*) [14], CORBA [4], ACE (*Adaptive Coomunication Environment)* [16] [17], PREMO [7] e A/V *Streams* [8]. Em especial, os *frameworks* PREMO e A/V *Streams* são fundamentalmente importantes no contexto desta tese, uma vez que muitas das idéias exploradas são baseadas em conceitos definidos nessas propostas. Vale a pena destacar que estes *frameworks* apresentam componentes com elevados níveis de flexibilidade. Propostas mais recentes envolvendo conceitos de metamodelos [18] [19] e reflexividade [20] também adotaram em suas definições o modelo de referência RM-ODP.

Concomitantemente, tem-se observado uma grande evolução na tecnologia de desenvolvimento de sistemas de *software* através da introdução do conceito de componentes. O princípio de uma

arquitetura baseada em componentes é flexibilizar a inclusão ou troca de componentes a partir dos requisitos e funcionalidade do sistema. Entretanto, esta tecnologia ainda não atingiu um nível de maturidade com relação aos aspectos inovadores demandados pelos sistemas multimídia baseados na *Internet* e, em especial, os sistemas de televisão digital interativa. Pode-se afirmar, porém, que o caminho a ser trilhado na busca de novas soluções inclui o modelo de desenvolvimento baseado em componentes. Esta tendência deve-se, entre outros fatores, aos níveis elevados de abstração do modelo, o que torna a programação menos complicada, e à possibilidade de reutilizar soluções configuráveis para tratar aspectos funcionais e não funcionais relacionados, por exemplo, a características tecnológicas específicas das plataformas, ou mesmo gerar automaticamente algumas partes da solução usando servidores de aplicações [21].

1.2 Componentes de *Software* e Adaptação

O interesse por mecanismos de suporte ao desenvolvimento de *software* adaptativo tem crescido muito nos últimos anos devido aos requisitos das aplicações atuais. Grande parte das abordagens publicadas nesse contexto usa a tecnologia de componentes [10] [22]. O tratamento da adaptação nessas abordagens normalmente está associado aos conceitos de configuração e reconfiguração. A configuração trata aspectos relacionados com a estrutura inicial e o controle dos elementos do sistema e dos componentes da aplicação. A reconfiguração está associada com as mudanças ocorridas durante a execução.

Um processo de adaptação, segundo McKinley [10], pode ser tratado usando duas possíveis abordagens de reconfiguração:

1) a primeira abordagem consiste em alterar o comportamento interno de componentes através de atualizações de valores de suas variáveis. Um exemplo deste tipo de adaptação freqüentemente citado na literatura é o protocolo TCP [23]. O protocolo ajusta seu comportamento trocando valores que controlam o gerenciamento de janelas de tempo para retransmissão em resposta a situações de aparente congestionamento da rede. Embora este tipo de adaptação não permita a troca dinâmica de algoritmos, nem a introdução de novos componentes após a construção da aplicação inicial, a definição de componentes configuráveis, associada ao uso de padrões de configuração, permite a realização de adaptações com níveis razoáveis de flexibilidade;

2) a segunda forma ocorre no nível da aplicação, introduzindo ou removendo componentes da configuração inicial. Este tipo de adaptação é mais flexível à medida que permite a utilização de componentes que podem ser montados e instalados dinamicamente, aumentando assim as funcionalidades do sistema implantado. Entretanto, o nível de complexidade associado à esta técnica é bastante elevado, principalmente no que se relaciona à definição do momento apropriado para realizar a troca dos componentes sem comprometer o andamento e a consistência do sistema. Assim, o componente que está sendo substituído não deve se encontrar no meio de uma operação. Em geral, para passar as responsabilidades do processamento de um componente original para um alternativo, deve-se transferir junto as variáveis de estado do ambiente.

O conceito de adaptação (auto-ajuste de código) é bastante antigo. Esta idéia já estava presente no projeto do Computador ENIAC ao explorar o conceito de programa armazenado de Von Neuman [24]. No modelo de Neuman, tanto instruções quanto dados podiam ser armazenados em memória. Isto permitia que instruções pudessem ser trocadas rapidamente de forma que novos programas pudessem ser executados, e mais ainda, que programas pudessem gerar novos programas.

A maioria dos trabalhos envolvendo adaptação com base em reconfiguração trata o processo através do uso de indireção. Através desta técnica, interações podem ser interceptadas para, dependendo do estado presente do sistema, realizar um eventual redirecionamento para um novo componente. Dependendo do estado, o suporte pode decidir, por exemplo, mudar o comportamento de alguns componentes, ou até mesmo trocá-los. Segundo McKinley [10] [25], as principais tecnologias exploradas atualmente para dar suporte a adaptação, são as seguintes:

- Separação de aspectos da computação por interesses (*concerns*);
- Computação reflexiva;
- Projeto baseado em componentes.

A técnica de separação da computação por interesses (*concerns*) tem sido tratada e explorada por inúmeros trabalhos, consistindo num importante princípio da Engenharia de *Software*. Esta técnica permite separar o desenvolvimento da aplicação com foco, por exemplo, em aspectos funcionais, e concentrando a análise no modelo de negócios e em códigos para interesse transversal (*crosscutting concerns*) tais como QoS e segurança. A separação de interesses transversais em relação aos aspectos funcionais da aplicação, além de simplificar o desenvolvimento e manutenção da aplicação, aumenta o

nível de reusabilidade. Atualmente, estas questões são suportadas e tratadas por abstrações incorporadas no modelo de Programação Orientada a Aspectos [26].

O conceito de computação reflexiva refere-se à habilidade de permitir que a aplicação possa consultar e eventualmente alterar o seu próprio comportamento. Estas idéias são abordadas por Szypersky [27] juntamente com os conceitos de reificação e metaprogramação. O conceito de reificação está associado à preservação de informações de compilação do componente de modo a permitir o seu uso dinamicamente pelo sistema. A técnica de suporte ao uso e modificação destas informações é chamada por Szypersky de metaprogramação.

A Fig. 1.1 mostra um modelo que explora o conceito de reflexividade. No plano inferior, a figura representa componentes de nível básico, enquanto que no plano superior, a figura mostra metacomponentes onde são descritas as propriedades dos respectivos componentes de nível básico.

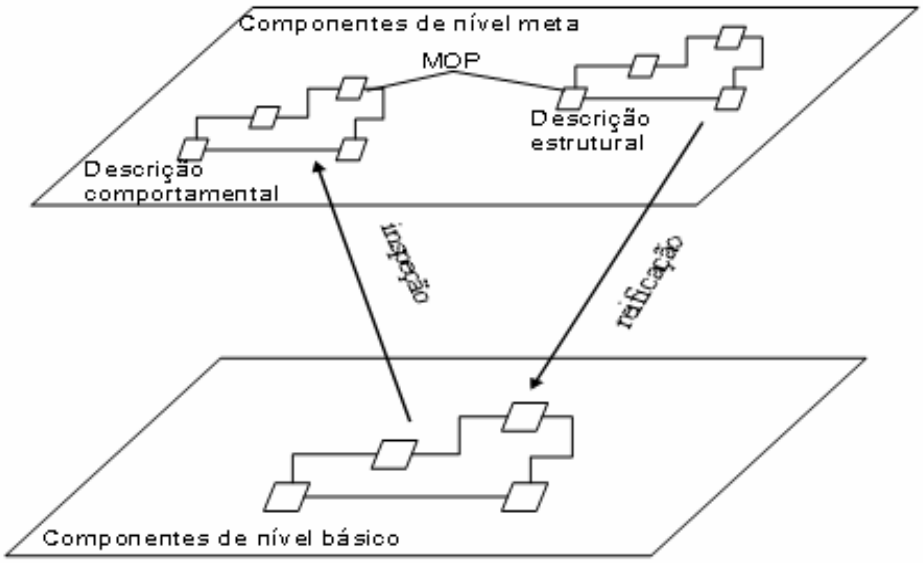


Fig. 1.1: Uma visão de sistemas reflexivos

Um modelo reflexivo normalmente envolve dois tipos de operação: introspecção, onde o sistema ou a própria aplicação inspecionam o seu comportamento, e reificação, onde alterações nos componentes básicos do sistema podem ser realizadas de acordo com as descrições nos respectivos metacomponentes.

O conceito de protocolo de metaobjetos (MOP - sigla oriunda do termo *Metaobject Protocol* em inglês) [29], consiste na definição de interfaces que permitem a realização de operações para introspecção e atualização de propriedades de metacomponentes.

Estas operações normalmente podem tratar aspectos estruturais, como por exemplo relação de dependência, hierarquia de classes e localização de componentes, assim como aspectos comportamentais, como o conjunto de interfaces providas por um componente e os protocolos de comunicação suportados.

Por fim, McKinley [10] destaca a técnica de Desenvolvimento Baseado em Componentes. O modelo de Desenvolvimento Baseado em Componentes de *Software*, apesar de ainda ser considerado como uma tecnologia recente da área de Engenharia de *Software*, tem se encaminhado para níveis de amadurecimento que viabilizam o seu uso em diversas áreas. A idéia consiste em usar esta tecnologia para construir programas adaptativos obedecendo a um modelo que dê suporte e consistência às operações de adaptação.

O nível de flexibilidade de um sistema poderá ser incrementado se o processo de montagem de componentes requeridos for retardado de modo a ser realizado em tempo de execução, sob demanda, à medida que as necessidades forem identificadas. Esta forma de abordagem incremental permite por exemplo, a inclusão de novas características e funcionalidades eventualmente não identificadas em tempo de projeto. Por outro lado, apesar desta abordagem proporcionar níveis altos de flexibilidade, retardar o processo poderá introduzir problemas relacionados ao gerenciamento da consistência da aplicação.

Na análise apresentada por Emmerich [30] são exploradas algumas idéias envolvendo a introdução de novos componentes ao modelo em tempo de execução. Nessa análise são mencionadas várias preocupações relacionadas com o nível de estabilidade da arquitetura para o tratamento consistente dos requisitos não-funcionais. Por exemplo, quando a composição utilizada na montagem do componente ocorre na fase de desenvolvimento, ou seja, em tempo de compilação, ou mesmo em tempo de implantação do componente, o dinamismo, apesar de limitado, é seguro, no sentido de que permite o sistema analisar a consistência da aplicação observando as características e restrições dos modelos de componentes e de configuração definidos previamente. Desta forma, uma solução de compromisso que oferece bons níveis de flexibilidade consiste em definir componentes com características configuráveis, onde os níveis de variabilidade dessas características são estabelecidos em tempo de projeto, com possibilidades de escolha de propriedades para estas características no momento da implantação ou da instanciação dos componentes, bem como dinamicamente, de acordo com as necessidades da aplicação.

1.3 Programação da Adaptação

Com o objetivo de facilitar o desenvolvimento de sistemas adaptativos, as plataformas de suporte precisam fornecer mecanismos e construções de alto-nível para especificar a programação de adaptações nesses sistemas. Assim, considerando uma análise envolvendo algumas propostas recentes, as seções seguintes discutem algumas questões que devem ser tratadas no suporte à adaptação, como por exemplo, onde colocar o código de adaptação, quando definir a programação da adaptação, quando realizar a adaptação e como programar a adaptação.

1.3.1 Onde Colocar o Código da Adaptação?

Um aspecto importante para o processo de adaptação consiste em definir onde deve ser introduzido o código de adaptação. Dependendo do local, pode-se comprometer o nível de flexibilidade. A análise apresentada por McKinley considera as seguintes possibilidades [10]: introduzir o código no *middleware* ou na própria aplicação. As vantagens e desvantagens associadas a cada uma delas são brevemente discutidas a seguir.

Código de adaptação no *middleware*

O mecanismo de gerenciamento de adaptação na camada de *middleware* geralmente envolve duas grandes abordagens:

- A primeira abordagem consiste em definir uma camada de serviços de comunicação adaptáveis. Esta técnica é usada na sub-camada de nível superior do *middleware* AdapTV [31], definido no escopo da presente tese, provendo funcionalidades para introspecção e modificação. Como outro exemplo, o *middleware* ACE [16] define uma camada *wrapper* que embute código para realização de *bindings* e dá suporte à realização de comunicação entre processos remotos. Ensemble [32] é outra proposta que também explora uma arquitetura em camadas e que permite a seleção de protocolos de comunicação específicos;
- A segunda abordagem consiste em prover uma máquina virtual com facilidades para interceptar e redirecionar interações envolvendo chamadas de funções. Por exemplo, JVM e CLR (*Common Language Runtime*) facilitam o processo de reconfiguração dinâmica através dos mecanismos de

reflexão fornecidos, respectivamente, pelas plataformas Java e .NET [33]. R-Java suporta metaobjetos através da adição de uma nova instrução ao interpretador Java. Uma abordagem desta natureza é muito flexível em relação à reconfiguração uma vez que permite a introdução de novo código, dinamicamente. No entanto, ela requer a especialização da máquina virtual de forma a prover transparência para a aplicação, o que reduz a portabilidade [10].

Ao introduzir funcionalidades de adaptação em sub-camadas de nível superior ao *middleware*, como por exemplo serviços de suporte à comunicação distribuída, provê-se suporte para portabilidade entre diferentes máquinas virtuais. Esta técnica envolve tipicamente componentes do *middleware* para interceptar chamadas a métodos remotos, redirecionando ou modificando o comportamento de acordo com as condições atuais [19]. Em alguns *frameworks*, o desenvolvedor estabelece explicitamente chamadas para serviços adaptativos do *middleware* [34] [35] [36]. QuO [9] usa *wrappers* para *stubs* e *skeletons* de CORBA de modo a obter o controle na chamada da operação, enquanto IRL [37] e ACT [38] usam interceptadores, os quais desempenham o papel de um gancho genérico (*hook)* que pode ser usado em tempo de execução para carregar outros tipos de interceptadores.

Código de Adaptação na Aplicação

O tratamento da adaptação no *middleware* provê níveis elevados de transparência, porém a abordagem se aplica somente aos programas desenvolvidos para aquela plataforma específica. Uma abordagem mais geral consiste em deixar que a aplicação realize a sua própria adaptação. McKinley considera duas possíveis técnicas [10] [25]:

- A primeira consiste em programar toda a aplicação, ou parte dela, usando uma linguagem que suporta diretamente o mecanismo de reconfiguração dinâmica. Algumas linguagens provêem mecanismos de reconfiguração dinâmica diretamente, como por exemplo, CLOS [39] e Python [40]; outras linguagens como Open Java [41], R-Java [42], PCL [43] e Adaptive Java[44], definidas como extensões da linguagem Java, incluem novas palavras reservadas e construções com o objetivo de prover capacidades para expressar código de adaptação. Entretanto, esta abordagem requer, por parte do desenvolvedor, o uso explícito destas construções para definição de seus programas;
- A segunda técnica consiste em combinar os aspectos funcionais e de adaptação no mesmo escopo. AspectJ [45] e Composition Filters [46] inserem, em tempo de compilação, código para

adaptação nos código das aplicações. Diferentemente, TRAP/J [47] usa uma abordagem em dois passos. No primeiro passo, um processo de combinação de aspectos insere, em tempo de compilação, ganchos de interceptação genéricos implementados como aspectos no código da aplicação. No segundo passo, realizado em tempo de execução, um processo de reconfiguração dinâmica pode introduzir novos componentes adaptativos na aplicação. Neste caso, um protocolo de metaobjetos usa operações de reflexividade para direcionar as operações interceptadas para os correspondentes componentes de adaptação. Esta estratégia permite adicionar novos comportamentos às aplicações existentes de forma totalmente transparente ao código original. Esta possibilidade é importante para permitir aplicações legadas executarem em infra-estruturas diversificadas.

1.3.2 Quando Definir a Programação da Adaptação?

Conforme foi mencionado na Seção 1.2, a possibilidade de introduzir ou remover dinamicamente componentes da configuração inicial aumenta o nível de flexibilidade do sistema. No entanto, a análise apresentada mostrou que retardar a programação, de modo a permitir a inclusão de componentes não conhecidos antecipadamente, requer muito esforço dinâmico para manutenção da consistência. Desta forma, uma boa alternativa para gerenciamento de consistência é permitir a realização de montagem e instalação dinâmica de novos componentes usando mecanismos de pré-programação dinâmica. Um esquema deste tipo, denominado adaptação antecipada, foi explorado por Cerqueira [48] onde são indicadas, entre outras informações, os componentes comuns e suas respectivas propriedades, assim como as condições de adaptação. Uma adaptação não-antecipada, denominada no presente texto como não-programada, supõe que a programação não conhece antecipadamente todos os componentes, ou seja, a questão da garantia da consistência neste caso se torna bastante crítica.

1.3.3 Quando Ativar a Adaptação?

Um dos elementos básicos utilizados pelos modelos de desenvolvimento baseados em componentes que dá suporte à adaptação é o conceito de composição. A maioria das abordagens analisadas consideram dois tipos de composição: estática e dinâmica [10] [17] [29] [30] [48] [49].

Na composição estática, um desenvolvedor pode combinar em tempo de compilação vários componentes para gerar uma aplicação. Na composição dinâmica, o desenvolvedor ou a própria aplicação pode adicionar, remover ou reconfigurar componentes em tempo de execução. Neste caso, é necessário que o ambiente suporte mecanismos de ligação tardia (*late binding*), permitindo assim, em tempo de execução, a realização de acoplamento de componentes compatíveis. As interfaces dos componentes funcionam como cláusulas de um contrato que especificam as responsabilidades das mesmas. A possibilidade de explorar a técnica de composição usando componentes desenvolvidos por terceiros, juntamente com o suporte à ligação tardia, aumenta o nível de reusabilidade e facilita o emprego da técnica de adaptação baseada no modelo composicional.

Layaida [50], Issarny [51] e Blair [52] apresentam abordagens de composição e configuração usando uma combinação das técnicas introduzidas, consistindo em:

- **Configuração estática sem reconfiguração.** Exemplos típicos desta abordagem definem elementos intermediadores dos tipos conversores e transcodificadores de fluxos multimídia [53] [54] [55] [56]. Embora estas propostas tenham tratado questões de heterogeneidade, elas empregam uma visão monolítica para projetos de aplicações. Ou seja, esta solução não suporta todas as possíveis situações geradas em decorrência da heterogeneidade do ambiente.

- **Configuração estática com reconfiguração estática.** Esta abordagem foi explorada nas ferramentas MBONE para suportar sistemas de conferência multimídia na *Internet* [57] [58] com base no protocolo RTCP [59]. A limitação desta proposta está relacionada com as políticas de reconfiguração, as quais são determinadas internamente no código da aplicação, ou seja, alterações envolvem um esforço considerável.

- **Configuração estática com reconfiguração dinâmica**. Nesta abordagem, as políticas de reconfiguração não são definidas no código da aplicação; elas são declaradas separadamente usando linguagens de configuração ou *scripts* [60]. Assim, *scripts* de reconfiguração são traduzidos em tempo de execução, no momento de ativação da aplicação. Nesta abordagem, o *parser* associado precisa ter conhecimento da arquitetura da aplicação – ou seja, os mecanismos de reconfiguração são dependentes da aplicação. O uso do mecanismo em outras classes de aplicações requer o desenvolvimento de um novo *parser*.

- **Configuração dinâmica sem reconfiguração**. Esta abordagem está presente em muitas arquiteturas de adaptação envolvendo propriedades de componentes de redes com o objetivo de

suportar diferentes requisitos. Por exemplo, Amir [61] descreve um *framework* usando arquiteturas baseadas em agentes onde uma aplicação cliente pode iniciar explicitamente um processo de adaptação em um dado *gateway*. Com objetivos similares, o *framework* Infopipes [62] permite configurar um processo de adaptação no nível da aplicação compondo separadamente os elementos definidos. Estas abordagens provêem muito mais flexibilidade para a configuração de aplicações, entretanto elas não tratam aspectos de reconfiguração.

- **Configuração dinâmica com reconfiguração estática**. Modelo usado pelo *framework* CANS (*Composable Adaptive Network Services*) [63] para a definição de uma arquitetura de adaptação envolvendo elementos da rede. Atividades de reconfiguração são usadas através de um conjunto de gerentes responsáveis por monitorar recursos e disparar reconfigurações quando for necessário. No entanto, as políticas de reconfiguração são amarradas ao código dos gerentes. Esta abordagem é limitada, uma vez que não permite a especialização das políticas em função dos requisitos da aplicação. Uma abordagem semelhante foi realizada por Morley [64] para adaptação de fluxos em aplicações multimídia móveis.
- **Configuração dinâmica com reconfiguração dinâmica.** Uma solução que explora este modelo foi apresentada por Rodriguez [65]. Este trabalho apresenta uma extensão da linguagem Lua [66] para dar suporte a configuração de aplicações baseadas em CORBA. Esta extensão inclui a definição de um *binding* baseado na interface DII do CORBA (*Dinamic Invocation Iterface*), permitindo componentes CORBA serem dinamicamente incorporados em aplicações. Devido ao fato de Lua ser uma linguagem interpretada, ela provê flexibilidade para a criação de código dinamicamente.

1.3.4 Como Programar a Adaptação?

Esta seção discute questões relacionadas com programação, configuração e adaptação de sistemas considerando requisitos e características de linguagens. Nesse contexto, a literatura apresenta algumas categorias de linguagens com estas finalidades, consistindo basicamente nas seguintes categorias:

- **Linguagens para interconexão de módulos** (*Module Interconnection Languages* - MILS) têm como objetivo descrever a estrutura de um sistema a partir de componentes previamente

implementados. MILS podem ser consideradas como linguagens para programação de granularidade grossa (*programming in the large*), cujo papel consiste basicamente em integrar componentes que foram previamente programados seguindo a técnica denominada *programming in the small*. MILS devem fornecer construções para identificar módulos de sistemas de *software* e para definir especificações para montagens de programas. O objetivo é prover características de flexibilidade, legibilidade e manutenibilidade. Especificações em linguagens MILS não estão preocupadas com questões relacionadas com as funcionalidades do sistema, ou seja, como as principais partes do sistema estão embutidas dentro da organização, ou mesmo como os módulos individuais implementam suas funções. Experiências relacionadas com esta visão de programação no contexto de sistemas distribuídos foram desenvolvidas no escopo do Sistema STER [67] e nas linguagens de programação LPM e LCM [68].

- **Linguagens de Descrição Arquitetural (ADLs)** são usadas para definir e modelar a arquitetura do sistema antes da implementação dos seus componentes. ADLs podem ser consideradas como ferramentas que dão suporte à programação de granularidade grossa (*programming in the large*), onde são descritos os elementos arquiteturais do sistema, como componentes, conexões, composição, propriedades não-funcionais, restrições e paradigmas de comunicação, entre outros. As ADLs normalmente apresentam construções e conceitos comuns em relação à maioria dos conceitos explorados nos modelos de desenvolvimento baseados em componentes [69].

- **Linguagens de Configuração** apresentam construções para definição de especificações envolvendo aspectos da estrutura e controle do sistema. A configuração de uma aplicação deve permitir a utilização de componentes que são potencialmente desenvolvidos por terceiros e de forma totalmente independente. Linguagens de configuração não se constituem numa nova abordagem; por exemplo, a linguagem Darwin [70] foi definida com esse propósito no contexto do ambiente Regis [71], provendo o conceito de composição e de relações do tipo requisição e provisão de serviços.

Uma técnica normalmente adotada para especificação de negociação em linguagens de configuração é definir contratos onde são indicadas obrigações e requisitos de componentes [72]. Devido à possibilidade de ocorrer incompatibilidades em uma composição entre componentes arquiteturais [73], a configuração deve tornar explícita certas características dos componentes, relacionadas por exemplo com o tipo do componente, o tipo da conexão, a arquitetura global, o

processo de construção e os requisitos não funcionais. Estes aspectos são tratados através de mecanismos de descrição de restrições (*constraints*). O uso de metaprogramação [27] permite a realização de consultas e inclusão dinâmica de novas informações estruturais do sistema, mecanismos fundamentais para a realização de negociações.

Com o objetivo de facilitar o desenvolvimento de sistemas grandes e complexos a arquitetura de *software* também define técnicas para descrição de sistemas de *software*. A arquitetura define o sistema como um conjunto de componentes claramente separados e as respectivas regras e restrições de interação. A definição de uma arquitetura de *software* é geralmente realizada nas primeiras etapas de um projeto, cujo objetivo consiste em conceber uma estrutura que atenda os requisitos funcionais e não-funcionais do sistema. Por outro lado, a tecnologia de componentes está mais preocupada com o processo de composição e configuração, envolvendo, entre outros aspectos, a seleção, a montagem e a distribuição de componentes em tempo de execução. A arquitetura constitui-se num meio para manter o conhecimento da estrutura da aplicação e do sistema durante a execução.

Linguagens de programação orientadas ao desenvolvimento baseado em componentes devem fornecer abstrações para tratar os conceitos relacionados com componentes de *software* e suporte para gerar e compor componentes. Prover suporte à programação de componentes no nível da linguagem facilita o gerenciamento estático da consistência em relação ao estabelecimento de restrições e negociações, desde que a linguagem seja aderente a um determinado modelo de referência.

Para prover os requisitos de um sistema, os componentes definidos podem ser considerados como de propósito específico - definidos especialmente para um sistema, componentes reusáveis - desenvolvidos para uso em vários sistemas da organização, ou componentes de uso comercial (COTS).

Para serem integrados, componentes pré-existentes eventualmente precisam de um "código de cola", ou algum nível de ajuste de comportamento do mesmo. Esta é uma abordagem *top-down* normalmente utilizada para ajustar o componente de forma a atender os requisitos da aplicação. Naturalmente que esta abordagem não estimula o uso de componentes pré-existentes, em especial quando se trata de componentes não-comerciais, pois existe uma grande probabilidade de que estes componentes não se adaptem ao contexto do sistema.

Outra abordagem consiste em iniciar analisando os requisitos do sistema para identificar os possíveis componentes candidatos que provêem suporte para estes requisitos. A especificação e seleção de componentes têm impactos nos requisitos finais e na arquitetura do sistema. Numa situação ideal, bastaria para o projetista de *software* identificar os componentes adequados ao sistema e descrever uma

especificação com a configuração dos mesmos. Desde que os artefatos usados pelas abordagens da Arquitetura de *Software* e da tecnologia de componentes envolvem os conceitos de componentes e composição, é natural que haja uma integração no uso de técnicas, ferramentas e métodos comuns. Assim, por exemplo, uma linguagem de descrição arquitetural pode ser usada como base para o projeto de um modelo de suporte à programação de sistemas baseados em componentes.

Nesta linha, Crnkovic [74] sugere considerar, para efeito de seleção de componentes, propriedades associadas às funcionalidades e a aspectos não-funcionais dos componentes. Assim, a escolha de um componente pré-existente envolve um processo de análise e negociação. Por exemplo, uma propriedade não-funcional identificada para um componente candidato poderia se enquadrar numa classificação que o considera como componente com alto-nível de reusabilidade; outro candidato à seleção poderia ser um componente que tem uma propriedade associada representando um nível baixo de eficiência. A seleção de componentes é desta forma um processo complexo devido a incertezas e ao não-determinismo inerente do ambiente. A seleção deve ser baseada nas propriedades definidas no modelo arquitetural considerando a compatibilidade com a arquitetura definida, a qual descreve as propriedades não- funcionais do sistema.

1.4 Motivação

Conforme foi introduzido nas seções anteriores, o desenvolvimento de sistemas multimídia distribuídos envolve muitos problemas. Recentemente, vários grupos de pesquisa e de desenvolvimento têm trabalhado e gerado propostas de novos modelos e plataformas de suporte para estes sistemas, porém, a cada dia surgem tipos diferentes de aplicações que apresentam novos requisitos.

O suporte à adaptação tem um papel fundamental neste contexto. A análise e o estudo realizados indicam que as soluções caminham na direção do uso de plataformas de *middleware* adaptativos e configuráveis explorando os conceitos de componentes e arquitetura de *software*. A abordagem de tratar os componentes do sistema com uma visão arquitetural simplifica o processo de desenvolvimento, uma vez que o foco fica delimitado para a descrição das propriedades e relacionamentos associados com a arquitetura. Entretanto, o desenvolvimento baseado em componentes e o conceito de modelos arquiteturais ainda são abordagens novas no contexto da Engenharia de *Software*. Algumas questões ainda precisam ser pesquisadas no sentido de construir plataformas de suporte ao uso dessas tecnologias. Em particular, os aspectos de configuração de componentes e

aplicações precisam ser aprimorados para facilitar a geração de aplicações envolvendo muitos componentes, os quais, por sua vez, podem ser construídos usando outros componentes.

A adaptação em sistemas distribuídos precisa preservar as características de cada componente individualmente, e ao mesmo tempo, tratar os requisitos gerais da aplicação considerando os relacionamentos e propriedades dos componentes distribuídos nas plataformas. Como componentes podem ser adquiridos de diferentes fornecedores, os quais certamente terão diferentes capacidades, é necessário prover mecanismos para dar suporte a negociações de propriedades e adaptações, de forma a atender os requisitos da aplicação. O problema é extremamente complicado devido à diversidade de abordagens envolvidas nas diversas camadas do *software*. Há incompatibilidades até mesmo para mecanismos definidos na mesma camada. Desta forma, há uma necessidade evidente de soluções que ofereçam facilidades para integração de componentes adaptativos num universo complexo que é constituído pelos sistemas multimídia distribuídos.

Sistemas adaptativos devem responder a esta dinamicidade de requisitos. Eles devem atuar de forma autônoma, negociando e modificando a composição do *software* para atender, da melhor forma possível, às necessidades impostas pelas aplicações, sem colocá-las em situações de risco ou interrupção. Várias pesquisas tratam estas questões; no entanto, a maioria das experiências relatadas foram desenvolvidas em domínios específicos [10]. A dinamicidade dos ambientes e a complexidade do *software* são elementos que dificultam o reuso das soluções em diferentes domínios.

Apesar dos grandes avanços ocorridos em relação às tecnologias de suporte à adaptação composicional, explorar estas potencialidades envolve inúmeros desafios. Por exemplo, o projeto de reconfiguração requer, por parte do paradigma de programação, o suporte automático à verificação das propriedades funcionais e não-funcionais do sistema.

Para prover consistência na realização de adaptação, o ambiente deve primeiramente certificar se todos os componentes apresentam as propriedades requeridas na especificação. Isto pode ser obtido selecionando componentes que tenham sido verificados e validados anteriormente através de técnicas tradicionais da engenharia de *software*.

Outra forma, pode ser através do uso de ambientes que geram automaticamente parte dos componentes a partir de especificações [21]. Esta abordagem foi explorada por Guimarães [19] no contexto de aplicações telemáticas.

Deve-se destacar, entretanto, que ambas abordagens têm limitações. Pode ser difícil encontrar componentes que atendam plenamente os requisitos da aplicação; por outro lado, tratar a geração de

novos componentes durante a execução pode levar a situações de instabilidade da arquitetura [30]. A presente tese optou por explorar a primeira abordagem. Apesar da limitação indicada, esta abordagem foi escolhida devido ao potencial que ela oferece para realização de adaptações dinâmicas seguras através de mecanismos do tipo adaptação pré-programada. Nesta abordagem, a negociação e seleção de potenciais componentes podem ser realizadas antecipadamente com base na especificação da aplicação; considera-se portanto que componentes podem ser especializados e ajustados por processo de configuração, de modo a atender às necessidades do sistema. Nesta direção, um modelo de gerenciamento e negociação de propriedades [31] [75] [76] [77] foi definido como uma evolução das propostas PREMO [7] e *A/VStreams* [8]. Paralelamente, considerando as experiências envolvidas no desenvolvimento desta tese, Borelli [78] avançou com novas contribuições na linha de negociação de propriedades propondo o BRICKS, que é um modelo de componentes com suporte à composição baseada em negociação de propriedades de interfaces.

1.5 Objetivo

Diante das motivações apresentadas, esta tese tem como objetivo conceber e desenvolver um *framework* de *middleware* (Cosmos) para configuração e gerenciamento de recursos e componentes da camada de *middleware* de sistemas multimídia distribuídos abertos. O conceito de *framework* normalmente está associado a técnicas de reuso no processo de desenvolvimento de *software* [79] [80]. Johnson [79] define um *framework* como um projeto reusável para todos, ou parte dos elementos de um sistema, sendo representado como um conjunto de classes abstratas e um conjunto de regras que descrevem as interações entre as instâncias destas classes. Outra definição explorada por Johnson considera um *framework* como um arcabouço de uma aplicação que pode ser configurado pelo desenvolvedor da aplicação. As duas visões são complementares onde a primeira define a estrutura de um *framework* e a segunda a sua função. Nos modelos de desenvolvimento baseados em componentes, o termo *framework* de componentes está associado também à definição de uma infra-estrutura de suporte à execução de componentes [27]. Estas visões estão contempladas na tese e, de acordo com o modelo definido, Cosmos pode ser considerado como um *framework* do tipo caixa-cinza [80].

Considerando o estado da arte em desenvolvimento de sistemas multimídia distribuídos, o projeto de um *framework* para este tipo de aplicação deve tratar o problema na camada de *middleware*. Atualmente, as plataformas de *middleware* para sistemas multimídia distribuídos gerenciam a complexidade destes sistemas usando os conceitos de objetos e componentes. Estas abordagens

fornecem várias facilidades, porém, ainda são limitadas em relação a alguns requisitos dos sistemas multimídia, como suporte para gerenciamento de fluxos e QoS (Seção 2.2). Assim, com o objetivo de superar estas limitações, o projeto do *framework* Cosmos identifica e considera os seguintes requisitos relacionados com plataformas de *middleware* para sistemas multimídia distribuídos [13] [16] [27][69], entre outros:

- conectividade;
- abordagem arquitetural;
- QoS;
- adaptação;
- reconfiguração dinâmica;
- reflexividade.

Para tratar estas questões, os seguintes objetivos específicos foram definidos no contexto da tese:

- Incorporar ao *framework* funcionalidades inovadoras e mecanismos de suporte à adaptação compatíveis com os requisitos das aplicações multimídia baseadas na *Internet* e dos Sistemas de Televisão Digital Interativa (TVDI). O processo de adequação e especialização do *framework* deverá ocorrer através da especificação de uma configuração;
- Implementar um protótipo do *framework* e realizar uma instanciação do mesmo em um *middleware* para sistemas de televisão digital interativa envolvendo os principais componentes da arquitetura. O objetivo da implementação consiste em analisar a arquitetura sob as capacidades relacionadas com a adaptação dinâmica da QoS em fluxos envolvendo a interconexão de componentes remotos, monitoramento e adaptação, ficando fora do escopo, as questões funcionais de sistemas de TVDI.

1.6 Visão Geral do *Framework* Proposto

Um *framework* de *software* permite a construção de modelos extensíveis e adaptáveis [79] [80]. A adoção desta técnica para a definição do Cosmos deve-se ao fato dela favorecer a construção de

sistemas flexíveis, os quais podem ser adaptados para atender aos requisitos de novos tipos de aplicações, com rapidez e facilidade, reduzindo assim o esforço para construção de novos sistemas.

O *framework* Cosmos [75] [81] [82] explora o modelo de desenvolvimento baseado em componentes, provendo assim facilidades para reuso e adaptação. A abordagem consiste em reduzir a complexidade de desenvolvimento ao tratar elementos complexos como unidades que têm altos níveis de reusabilidade, ao mesmo tempo que esconde os aspectos de controle interno. O *framework* contempla os principais elementos do modelo definido no Cosmos, como por exemplo, a representação e gerenciamento de recursos, propriedades, QoS e configuração. O *framework* define uma arquitetura geral de *software* para projeto e desenvolvimento de sistemas de *middleware* a partir da integração destes elementos, permitindo, inclusive, a extensão dos mesmos. Com base no modelo definido no Cosmos, o projeto de um sistema desvincula as várias visões relacionadas a aspectos funcionais, não-funcionais, estruturais, de tecnologias e de especificação de aplicações, entre outros.

Esta abordagem de desacoplar diferentes aspectos das aplicações contribui também para a provisão de características de flexibilidade, permitindo que a instanciação do *framework* em plataformas específicas de *middleware* trate e especialize cada aspecto individualmente. Por outro lado, o *framework* permite reusar componentes que tratam aspectos básicos relacionados, por exemplo, com a representação e gerenciamento de recursos, propriedades, QoS e configuração. Devido ao uso da tecnologia de componentes, o *framework* permite introduzir, trocar ou reconectar componentes do sistema, simplesmente analisando as propriedades dos componentes.

A arquitetura definida estabelece que uma aplicação pode ser criada através de um processo de configuração, onde componentes são buscados, criados, customizados e eventualmente conectados a outros componentes. A adaptação ocorre através da configuração, reconfiguração e ajuste de propriedades dos componentes do *middleware* e da aplicação.

O *framework* define um modelo básico de componentes e um modelo de metacomponentes que serve de base para especificação de configuração e para descrição de componentes do *middleware* e de aplicações. Este modelo de metacomponentes contempla os elementos arquiteturais do *framework*, de forma que uma linguagem de especificação para o Cosmos tem em sua essência as características de uma ADL (*Architectural Description Language*). Neste modelo de especificação, aplicações e componentes podem ser especificados, construídos e configurados a partir do uso de outros componentes previamente desenvolvidos, através de técnicas de composição.

Os processos de gerenciamento de configuração e adaptação do *middleware* e das aplicações envolvem tanto aspectos definidos estaticamente, como aspectos que somente serão conhecidos dinamicamente, ou seja, durante a execução da aplicação. A reconfiguração dinâmica leva em considera mecanismos de pré-programação, quando são estabelecidas as diretrizes para a verificação da consistência em um processo de adaptação, assim como análises dinâmicas envolvendo inspeção dos estados da plataforma e dos componentes da aplicação.

Os conceitos de reflexividade e de ciclo de vida estão também presentes no Cosmos, constituindo-se em elementos fundamentais para o suporte ao gerenciamento de componentes da aplicação e dos recursos do sistema.

No Cosmos, a reflexividade é baseada na definição de metacomponentes utilizados para descrever e representar os requisitos dos componentes da aplicação e do *middleware*. Metacomponentes são suportados e gerenciados usando o conceito de propriedades [75]. O principal propósito dos metacomponentes consiste em prover metainformações para estabelecimento de acordos dinâmicos considerando as capacidades das plataformas, as propriedades dos recursos, os formatos de dados manipulados pelos recursos e as questões de QoS envolvidas em processos de captura, processamento, transmissão e exibição de mídia num conjunto diversificado de elementos. Para resolver conflitos (idealmente, de forma automática) e para gerenciar o processo, Cosmos define um componente central denominado Configurador.

Como prova de conceito, o trabalho fez uma validação experimental do *framework* Cosmos explorando os modelos de QoS [83] [84] e interconexão de componentes [85] [86] [87]. Esta validação foi realizada no contexto de um protótipo de um *middleware* para sistemas de televisão digital interativa [31] [77] desenvolvido explorando os aspectos de configuração e gerenciamento de recursos definidos no Cosmos. Adicionalmente, o trabalho desenvolveu também uma aplicação exemplo envolvendo a captura, transmissão e apresentação de fluxo multimídia em uma plataforma distribuída. Nesta implementação foram especificados requisitos de QoS com o objetivo de testar os mecanismos de adaptação pró-ativa e reativa do Cosmos.

Deve-se destacar que as plataformas de *middleware* atuais para sistemas de televisão digital interativa não abordam questões de adaptação [88]. Entre outros requisitos para sistema TVDI, a análise apresentada em [88] destaca como essenciais os seguintes: reconfigurabilidade, adaptabilidade, reflexibilidade, ubiqüidade e mobilidade. Neste sentido, o presente trabalho foca o seu esforço no tratamento da adaptação, onde são considerados conceitos de *middleware* adaptativos [11] [89]. Em

adição, sistemas de TVDI provêem um ambiente adequado para validar a proposta, pois nestes sistemas a complexidade é ampliada em relação aos sistemas multimídia tradicionais em razão das novas tecnologias que serão incorporadas.

1.7 Principais Contribuições

A literatura apresenta uma diversidade de abordagens flexíveis e interessantes que exploram modelos de desenvolvimento baseados em componentes, principalmente em aplicações direcionadas para sistemas de informações corporativos. Nestas abordagens, o uso de servidores de aplicação tem sido vastamente explorado. No entanto, na área de sistemas multimídia distribuídos, onde requisitos temporais e de QoS, além de rígidos, apresentam variações intensas, a maioria das soluções foram experimentadas em sub-áreas específicas. Nesse sentido, características relacionadas à adaptação e QoS ainda precisam ser melhor investigadas, de forma a permitir a utilização delas adequadamente neste novo contexto de aplicações. Como princípio, novas soluções devem preservar requisitos essenciais como abstração e reusabilidade. Este trabalho se agrega ao esforço da comunidade, gerando as seguintes contribuições:

- O *framework* Cosmos, que provê uma arquitetura genérica da camada de *middleware* para gerenciamento e configuração de recursos e componentes de sistemas multimídia distribuídos abertos. O *framework* explora intensamente o conceito de propriedades na definição de metacomponentes, os quais dão suporte às características de reflexividade e adaptação dinâmica, tanto para componentes do *middleware*, como para componentes da aplicação;
- Um modelo de metacomponentes que dá suporte à adaptação, interoperabilidade e conectividade entre componentes desenvolvidos de forma independente de tecnologia. Este modelo funciona como base para definição de linguagens para descrição e configuração de aplicações e componentes;
- Um modelo de gerenciamento de QoS que dá suporte à especificação e gerenciamento de uma variabilidade de parâmetros de QoS, no nível de configuração de aplicações;
- Um modelo genérico de interconexão de componentes que permite tratar, de forma totalmente transparente para a aplicação, questões relacionadas com distribuição, tecnologias, topologias e tipos de protocolos;

- Validação experimental da camada de gerenciamento e configuração de um *middleware* para sistemas TVDI que foi projetado com base no *framework* Cosmos. Esta validação foi realizada como prova de conceito, correspondendo às implementações de um protótipo do *middleware* no contexto de um sistema específico e de uma aplicação exemplo.
- Definição de um modelo de configuração e reconfiguração de aplicações independente de tecnologia. A abordagem consistiu em definir um modelo de metacomponentes para descrição de componentes do *framework* e da aplicação. Uma instanciação do modelo foi realizada usando a tecnologia XML e utilizada para descrever a aplicação exemplo desenvolvida.
- Uma linguagem de especificação de configuração de aplicações para uma plataforma de *middleware* de TVDI baseada no Cosmos.

Para produzir estes resultados, o presente trabalho realizou atividades relacionadas com pesquisa, estudo, análise e desenvolvimento, gerando as seguintes publicações:

- Um ciclo de vida estendido para o Modelo ISO/PREMO e sua implementação em ambiente CORBA [90];
- Um modelo para especificação formal da configuração de aplicações baseadas no PREMO [91];
- Especificação Formal de um Modelo de Sincronização Baseado no PREMO[92];
- Uma plataforma para programação multimídia baseada no modelo ISO/PREMO e sua Implementação [93].
- COSMOS: Um *Framework* para Configuração e Gerenciamento de Sistemas Distribuídos Abertos [81];
- "Exploring an Open, Distributed Multimedia Framework to Design and Develop an Adaptive Middleware for Interactive Digital Television Systems" [31];
- "A Component-Based Configuration Framework for Open, Distributed Multimedia Systems" [75];
- "A XML-Based Component Specification Model for an Adaptive Middleware of Interactive Digital Television Systems" [76];
- Projeto e Implementação de um *Middleware* para sistemas de Televisão Digital Interativa [77];
- Uma Arquitetura para Configuração e Gerenciamento de Recursos em um *Middleware* para Sistemas de Televisão Digital Interativa [82];

- Especificação e Gerenciamento de QoS em um *Middleware* para Sistemas de Televisão Digital Interativa [83];
- "QoS Specification and Management in a Middleware for Distributed Multimedia Systems" [84];
- Uma Arquitetura de Interconexão de Componentes para Sistemas Multimídia Distribuídos [85];
- Um Modelo de Interconexão de Componentes e sua Implementação em um *Middleware* para Sistemas de Televisão Digital Interativa [86];
- "A Component Interconnection Model for Interactive Digital Television" [87].

1.8 Estrutura da Tese

A presente tese está organizada na forma descrita a seguir. O Capítulo 2 aborda os principais aspectos relacionados com a tecnologia atual de *middleware* e estabelece os principais requisitos no contexto dos sistemas multimídia distribuídos. Em seguida, o Capítulo 3 faz uma análise sobre o estado da arte onde são apresentados e discutidos alguns trabalhos e soluções relacionadas aos requisitos identificados no Capítulo 2. O Capítulo 4 apresenta o *framework* Cosmos e descreve os modelos arquitetural e funcional do mesmo, bem como a arquitetura para suportar o gerenciamento de QoS e de interconexão entre componentes do *framework*. Neste capítulo, são estabelecidas associações entre as interfaces funcionais definidas no Cosmos e o suporte aos requisitos indicados no Capítulo 2. Em seguida, o Capítulo 5 apresenta o *middleware* AdapTV – um *middleware* para sistemas de televisão digital interativa, baseado no Cosmos -, e uma descrição da implementação desse *middleware* realizada como prova de conceito. Por fim, o Capítulo 6 apresenta algumas considerações finais envolvendo uma avaliação qualitativa do *framework* proposto e uma análise comparativa com os principais trabalhos relacionados, concluindo com as perspectivas futuras.

Na apresentação do *framework* Cosmos, o texto destaca os principais conceitos relacionados. Para isto, no desenvolvimento do texto são apresentadas várias especificações envolvendo as APIs dos principais componentes, bem como a descrição de algumas classes que foram implementadas no protótipo do *middleware* AdapTV. A representação adotada para os conceitos de componentes e as respectivas dependências foi a notação definida em [94]. Para representação dos elementos das arquiteturas, uma notação baseada em UML foi explorada intensivamente. A nomenclatura adotada

nestas APIs e, por conseguinte, também no texto, utiliza em grande parte termos na língua inglesa. Isto foi adotado com o objetivo de uniformizar as linguagens utilizadas para a especificação e implementação.

Capítulo 2

Middleware para Sistemas Multimídia Distribuídos

O presente capítulo discute a importância das tecnologias de *middleware* no contexto de desenvolvimento de sistemas multimídia distribuídos. O capítulo inicia apresentando uma visão geral com as características de um sistema multimídia distribuído destacando que as soluções para estes sistemas envolvem níveis elevados de complexidade. Apesar do texto indicar que o uso de tecnologia de *middleware* pode reduzir o nível de complexidade associado ao desenvolvimento desses sistemas, a análise apresentada mostra algumas limitações das arquiteturas de *middleware* atuais com relação ao suporte à adaptação. Para finalizar, o capítulo apresenta alguns requisitos que foram identificados com o objetivo de dar suporte ao projeto e construção de plataformas de *middleware* adaptativas.

2.1 Características dos Sistemas Multimídia Distribuídos

Uma das principais características das aplicações multimídia que as diferenciam das aplicações tradicionais é o processamento integrado de múltiplas mídias. Em sistemas multimídia, além de texto, a informação pode ser representada através de áudio, vídeo, imagem, gráficos e animações [95].

Sistemas multimídia distribuídos são caracterizados por utilizarem computadores distribuídos ao longo de uma rede, implicando dessa forma em tráfego de informações multimídia. Esses sistemas envolvem tipicamente recursos específicos de *hardware* (placas de captura de áudio e vídeo, por exemplo), bem como *software* para codificação, processamento, transmissão, decodificação e apresentação de dados de mídia.

Estes sistemas são complexos porque, além dos aspectos temporais envolvidos com as mídias áudio e vídeo das aplicações, deve-se considerar que elas poderão ser executadas em diversos tipos de dispositivos, com capacidades variadas de processamento e armazenamento, e diferentes interfaces com o usuário. Este fato impõe a adoção de diferentes interfaces de interação, dependendo do tipo de dispositivo e da tecnologia de acesso empregada. Como exemplo, podem-se citar os *set-top boxes*, que

são dispositivos atualmente utilizados para receber e decodificar sinais de TV. Estes dispositivos incorporam conceitos e mecanismos dos sistemas operacionais modernos, provendo assim interfaces para uso de serviços comuns como acesso à memória, discos rígidos e interfaces de rede. Desta forma, em adição aos serviços de televisão convencional, estes dispositivos permitem a exibição de conteúdos de mídia armazenada localmente, ou a execução de programas distribuídos juntamente com o fluxo multimídia. Com estas facilidades, um telespectador pode, por exemplo, realizar interações com a estação produtora / transmissora de conteúdo de TV, de forma a estabelecer um canal de interatividade.

Outro fato que merece destaque é a atual evolução da computação ubíqua com o crescimento no uso de dispositivos móveis, como PDAs e telefones celulares. Estes dispositivos atualmente têm capacidade de processamento e resolução que suportam o processamento de aplicações multimídia. E mais ainda, eles também oferecerem interfaces que provêem suporte para conectividade à rede.

Apesar deste cenário heterogêneo e diversificado, pode-se constatar que as atuais aplicações multimídia ainda são construídas para serem executadas em plataformas e arquiteturas específicas [11]. Neste contexto, o princípio da reusabilidade se limita à construção de aplicações que manipulam dispositivos e formatos de dados específicos. Muitas das soluções ainda são construídas usando uma abordagem centralizada, onde o processamento se restringe ao escopo da plataforma e do sistema. A rede normalmente é utilizada apenas para transmissão de dados, sem obviamente explorar o grande potencial que ela oferece como fonte de recursos e componentes que proliferam rapidamente.

Mídias digitais normalmente requerem recursos computacionais mais poderosos com relação à capacidade de processamento, armazenamento e transmissão. Mesmo utilizando dispositivos para compressão de dados, ainda assim a demanda de banda passante em um sistema multimídia não é atendida por grande parte das tecnologias de rede atuais. Por outro lado, dispositivos móveis com pouca capacidade de processamento poderiam ser beneficiados através da utilização de recursos distribuídos na rede, que poderiam fazer a conversão do fluxo original para um formato compatível com as suas capacidades utilizando técnicas de adaptação executadas no servidor [96].

A indisponibilidade de alguns recursos no sistema, temporária ou não, em função do compartilhamento ou sobrecarga da rede, pode causar dificuldades em manter a qualidade de serviço exigida pela aplicação. Este problema é agravado na *Internet* devido, principalmente, ao não-determinismo intrínseco da rede. Há várias tentativas para tratar estas questões [97], porém o que se

percebe na prática é que muitas das soluções manipulam estes problemas olhando e atuando nas extremidades dos fluxos [98] [99].

A qualidade de serviço está portanto fortemente relacionada com políticas de gerenciamento de recursos [7] [100]. Os recursos necessários à execução de serviços multimídia envolvem tipicamente dispositivos de captura e exibição, bem como recursos de sistema (memória, CPU, rede, etc.) que normalmente são interconectados adotando um estilo arquitetural do tipo *pipe-filter* [7] [101]. É possível que estes recursos sejam compartilhados, o que pode ocasionar situações de conflito. Cabe à política de gerenciamento de recursos antecipar-se e solucionar essas situações. É tarefa do sistema operacional fornecer mecanismos para gerenciamento de recursos, porém esta abordagem pode ser considerada de baixo nível. A gerência sobre esses recursos pode, no entanto, ser tratada em um nível mais alto de abstração através de serviços de *middleware* (camada de *software* entre o sistema operacional e a aplicação) [102]. Tais serviços propiciam à aplicação simplicidade e transparência no uso e compartilhamento de recursos. A representação lógica de um recurso do sistema é freqüentemente denominada no *middleware* como recurso virtual [7] [8] [100], ao passo que a interconexão entre recursos, por conexão virtual [7].

A caracterização dos requisitos de QoS de uma aplicação depende das características das mídias envolvidas e dos requisitos dos usuários. É natural que exista uma variação inerente à sensibilidade de cada indivíduo. No entanto, para a execução de um serviço de qualidade, existe um limite mínimo de requisitos de apresentação cuja não obediência causa desconforto a qualquer indivíduo, bem como um limite superior a partir do qual é indiferente [103]. Desta forma, o gerenciamento de QoS deve levar em conta esses dois critérios. Além disso, ele precisa verificar se há disponibilidade dos recursos necessários à execução dentro desses limites, o que nem sempre é possível, uma vez que os sistemas multimídia distribuídos estão sujeitos a sobrecargas, falhas na rede e à própria capacidade dos recursos no local de destino.

Dessa forma, para prover flexibilidade, é comum oferecer um grau de liberdade na seleção de QoS nos sistemas multimídia distribuídos [9], cujo nível de negociação está entre os requisitos do cliente, as restrições impostas pela aplicação e a disponibilidade de recursos da plataforma.

Políticas para gerenciamento de recursos envolvem aspectos estáticos, como especificação de QoS, negociação para escolha de formatos para codificação/decodificação de mídia e controle de admissão, bem como aspectos dinâmicos relacionados com monitoramento, manutenção e renegociação de QoS, para fins de adaptação.

A especificação de QoS tem como função descrever os requisitos de QoS da aplicação, ou seja, uma descrição tem um papel análogo ao de um contrato que rege as normas de configuração e operação da aplicação.

A negociação tem como objetivo selecionar valores apropriados para os parâmetros de QoS que possam cumprir os requisitos da especificação.

O controle de admissão tem a finalidade de determinar se o sistema pode atender uma requisição de recursos em um determinado tempo. Logo após a verificação da viabilidade de uma requisição, o sistema deve realizar a reserva e aquisição efetiva do correspondente recurso.

O monitoramento de QoS consiste em verificar, em tempo de execução, se os recursos alocados à aplicação estão compatíveis com o contrato, ou seja, se os valores dos parâmetros de QoS estão dentro dos requisitos estabelecidos na especificação, relatando eventuais violações. Caso ocorram violações, as discrepâncias em relação ao contrato especificado devem ser consideradas pelo processo responsável por realizar a adaptação.

Outra questão que gera complexidade no desenvolvimento de sistemas multimídia é a sincronização de fluxos [95]. O problema consiste em assegurar que os dados de cada fluxo trafeguem numa cadência tal que possam ser exibidos no tempo esperado (sincronização *intrastream*), e mais ainda, que sejam sincronizados fluxos inter-relacionados (sincronização *interstream*). Desta forma, a sincronização neste contexto envolve relações temporais entre dados de mídia contínua. Na sincronização *intrastream*, a relação temporal está associada com os elementos do próprio fluxo, como por exemplo, os quadros de um fluxo de vídeo. Normalmente, esta relação é expressa em número de quadros por segundo, implicando em manter a taxa e reduzir o *jitter* (variação estatística do retardo). Na sincronização *interstream* a relação temporal estabelece uma vinculação entre elementos de diferentes fluxos com o objetivo de minimizar o deslocamento entre referências temporais associadas. Um exemplo clássico consiste na sincronização labial envolvendo fluxos de áudio e vídeo. Blakowsky [95] apresenta uma tabela com parâmetros indicando níveis de discrepâncias aceitáveis para sincronização *interstreams* envolvendo elementos das principais mídias.

A sincronização em geral envolve duas fases: a primeira normalmente é estabelecida de forma natural, no processo de captura, ou explicitamente quando os dados de mídia forem produzidos em um processo de edição. Na segunda fase, as relações temporais são usadas para restaurar a sincronização durante a apresentação dos dados da mídia. Desta forma, para realizar a sincronização, há necessidade de introduzir o conceito de relógio para gerar marcas de tempo (*timestamps*) associados aos dados de

mídia. O gerenciamento da sincronização consiste em observar os respectivos *timestamps* e tentar manter as relações originais, podendo ser realizado em várias camadas, envolvendo diferentes graus de precisão. Tanto na sincronização *intrastream*, quanto na *interstream*, os respectivos *timestamps* denotarão a correção e a QoS do fluxo. Para ajustar a sincronização, técnicas como descartar quadros podem ser aplicadas [93], porém alguns cuidados especiais devem ser tomados; por exemplo, em quadros de um GOP (*Group of Pictures*) de um fluxo de vídeo MPEG [104], o descarte de um quadro I faria com que os quadros P e B não fossem decodificados corretamente.

Visando diminuir a complexidade dos problemas discutidos, várias soluções têm sido propostas na literatura definindo plataformas na camada do *middleware*. As seções seguintes apresentam uma análise envolvendo as tecnologias empregadas nas arquiteturas de *middleware* atuais.

2.2 Arquiteturas de *Middleware*

Para tratar a complexidade envolvida no processo de desenvolvimento, gerenciamento e adaptação de sistemas multimídia distribuídos, vários esforços têm sido empreendidos nos últimos anos na camada de *middleware*. Inicialmente, as preocupações de projetos de *middleware* estavam relacionadas com o suporte à distribuição. Entretanto, as tecnologias atuais de *hardware* e *software* têm evoluído muito rapidamente, de modo que novos requisitos de sistemas e aplicações precisam ser considerados também na camada de *middleware*. Assim, o desafio consiste em conceber soluções de *middleware* flexíveis e adaptativas que possam incorporar novos requisitos. Nessa direção, a literatura tem mostrado uma tendência para definições de arquiteturas baseadas em componentes de *software* associadas aos conceitos de reflexividade, conforme pode ser observado nas discussões seguintes.

2.2.1 Arquiteturas Baseadas em Objetos

Um conceito fundamental presente na maioria das plataformas de *middleware* iniciais é o de orientação a objetos. As abordagens baseadas nos conceitos de objetos surgiram no contexto de plataformas distribuídas, com os seguintes objetivos:

- Facilitar o processo de desenvolvimento de sistemas distribuídos através da definição de abstrações de níveis mais altos que embutem consistência e ocultam detalhes específicos, como por exemplo, interfaces de acesso a redes e a serviços de sistemas operacionais, simplificando desta forma os requisitos da aplicação.

- Estabelecer diretrizes para o processo de desenvolvimento envolvendo aspectos de interoperabilidade e interconexão entre componentes do sistema.
- Permitir e simplificar o processo de integração de componentes desenvolvidos por diferentes fornecedores de tecnologias de *hardware* e *software*.
- Coordenar o processo de criação e execução de aplicações.

Nesse contexto, as abordagens CORBA[8], DCOM[5], Java RMI [6] e ACE [16] merecem destaque pelo papel que desempenham em relação à maioria dos fundamentos que são explorados nas soluções atuais. A preocupação dessas abordagens em ocultar detalhes específicos de arquitetura permite tratar, de maneira simplificada e uniforme, a heterogeneidade dos possíveis ambientes (dispositivos, sistemas operacionais e redes). Esta foi uma das principais preocupações que motivou o desenvolvimento do mecanismo de chamadas remotas de procedimentos (RPC) [105] – a tecnologia precursora do conceito de *middleware*. Vale a pena destacar que esta preocupação está presente em várias propostas como o PREMO [7], *A/V Streams* [8] e *Directshow* [106].

O uso dos conceitos de objetos na construção de aplicações multimídia distribuídas incorpora as características do modelo cliente-servidor. No entanto, o desenvolvimento de aplicações para manipulação de fluxos de mídia usando o modelo cliente-servidor tem limitações uma vez que o controle ocorre em um nível muito alto [107]; clientes e servidores são aplicações monolíticas, geralmente desenvolvidas para um único cenário, não podendo ser facilmente modificadas. Usando esta abordagem, haveria uma certa dificuldade, por exemplo, para distribuir um processo de produção de conteúdo de TV de um servidor centralizado para diferentes estações geradoras e transmissoras de conteúdo, especialmente se cada estação utilizar em seus estúdios tecnologias e formatos de codificação diversificados. Assim, mecanismos de adaptação dinâmica são elementos essenciais para plataformas de *middleware* darem suporte ao desenvolvimento e execução destes sistemas [107] [108]. Outras limitações citadas na literatura são listadas a seguir:

- **Mecanismos de interação inadequados para transmissão de fluxos de mídia contínua**: o estilo básico de interação do modelo de objetos consiste em operações do tipo *request-reply*, não sendo semanticamente adequado para lidar com os requisitos de QoS relacionados com o gerenciamento de transmissão de fluxo de mídia contínua [12].
- **Ausência de um mecanismo de estruturação clara dos aspectos funcionais**: segundo a análise de Wang [107], plataformas de *middleware* baseadas em objetos tratam as interfaces no contexto de contratos em que objetos desempenham papéis como cliente ou servidor. Nestas

abordagens, não há mecanismos para desacoplar as dependências relacionadas entre os objetos que colaboram nas implementações de outros objetos. Ou seja, se houver a necessidade de usar uma implementação provida por outro objeto, requer-se por parte da implementação a explicitação desta conexão, criando um vínculo de acoplamento forte. Assim, na programação de uma aplicação distribuída complexa, os desenvolvedores da aplicação devem identificar e interconectar todos os serviços e suas respectivas dependências, o que pode provocar um baixo nível de reusabilidade.

- **Ausência do conceito de servidores de componentes genéricos:** segundo Wang [107], sistemas de *middleware* baseados em objetos não provêem *frameworks* para realização de tarefas gerais como incializar o servidor e as políticas de QoS, nem tampouco mecanismos de gerenciamento em tempo de execução para cada componente. Este fato conduz a implementação *ad-hoc* de servidores com altos níveis de dependência com a plataforma e a aplicação, levando ao aumento da complexidade da evolução.
- **Dificuldades para manter a consistência em um processo de decomposição *top-down***: segundo Zhang [109], a abordagem de decomposição *top-down*, tradicional em plataformas de *middleware* que precisam mudar dinamicamente a estrutura de sua arquitetura, é inadequada. Como normalmente a modularização de um projeto neste contexto envolve o tratamento de requisitos ortogonais, o nível de complexidade associado se torna muito elevado, uma vez que a lógica de tratamento de um requisito pode afetar as lógicas de tratamento dos demais. Desta forma, usando modelos tradicionais, seria difícil manter a arquitetura consistente em relação a abstração inicial de alto nível. Assim, fica praticamente impossível desacoplar as várias abordagens, principalmente quando os requisitos associados ao domínio do *middleware* podem mudar freqüentemente.

2.2.2 Arquiteturas Baseadas em Componentes

Uma abordagem empregada recentemente para superar as limitações citadas em relação à tecnologia de orientação a objetos é a adoção do modelo de desenvolvimento baseado em componentes. O modelo de componentes permite a criação de uma fronteira virtual envolvendo os componentes da aplicação; componentes interagem com outros componentes através de interfaces bem definidas e relacionadas por critérios de dependência. Em adição, existem atualmente várias

abordagens que definem mecanismos e padrões para composição e execução de componentes em servidores de componentes genéricos, como, por exemplo o modelo CCM [21] [110].

Componentes são entidades de implementação que exportam um conjunto de interfaces para que outros componentes possam usar suas funcionalidades. Componentes podem também indicar que precisam (requisitam) de funcionalidades que serão providas por outros componentes, definidos e implementados fora de seu escopo.

Tecnologias de suporte ao conceito de componentes, como por exemplo o CCM [110], definem um mecanismo denominado *Contêiner* para prover um ambiente de suporte em tempo de execução para os componentes, contemplando por exemplo, mecanismos para notificação e tratamento de eventos, transação e segurança, entre outros.

A popularização da tecnologia de componentes através da definição de alguns modelos como CCM [21] [110], .NET [111] e EJB [112], tem permitido aumentar a escala de aplicações distribuídas através das facilidades de uso de componentes prontos, tipo COTS. Estes componentes são distribuídos em módulos que podem ser implantados em diferentes plataformas computacionais. *Frameworks* geradores de aplicação como o J2EE [113] e JBoss [114] podem ser adquiridos para dar suporte ao gerenciamento de componentes.

Embora o reuso de componentes básicos seja importante, o maior benefício do modelo está na possibilidade de construir componentes de nível mais alto utilizando outros componentes através de mecanismos de composição, e em última instância, na composição de aplicações.

As técnicas de composição ainda não estão maduras como nos sistemas de produção e montagem de componentes de *hardware*. Entretanto, a meta idealizada consiste em buscar uma aproximação em que o processo de desenvolvimento de sistemas de *software* complexos possa ser realizado através de técnicas de composição utilizando e configurando componentes pré-existentes [115].

Os principais papéis relacionados com as etapas que compõem o ciclo de vida de desenvolvimento baseado no uso de componentes, resumem-se nos seguintes:

- **Projetistas de componentes** - entidades responsáveis por definir as características dos componentes; seu papel de atuação está voltado para o nível de descrição das interfaces dos componentes.
- **Implementadores de componentes** - entidades responsáveis pelas implementações dos componentes.

- **Empacotadores de componentes** - entidades responsáveis por empacotar implementações de componentes observando suas respectivas propriedades.
- **Montadores de componentes** - elementos responsáveis pela seleção das implementações e composição dos mesmos em aplicações.
- **Configuradores** (*Deployers*): correspondem às entidades responsáveis pela distribuição, instalação e configuração de montagens de componentes em servidores de componentes.

Sistemas distribuídos de larga abrangência requerem a integração de muitos componentes de *hardware* e de *software*, normalmente desenvolvidos de forma independente. Apesar da tecnologia de componentes ter atingido um bom nível de maturidade para uso em larga escala em sistemas comerciais, a tecnologia não está suficientemente madura para suportar o desenvolvimento de sistemas em domínios que envolvem a interconexão de componentes distribuídos com restrições temporais críticas e de QoS [115]. Nestes domínios, o desenvolvimento constitui-se ainda numa tarefa complexa. Cabe então ao *middleware* reduzir o esforço de desenvolvimento do *software* através da utilização de técnicas de composição que favoreçam a reusabilidade nestes domínios. As plataformas atuais de *middleware* convencionais falham ao proverem abstrações para suporte de QoS que forçam os desenvolvedores controlarem e assegurarem a QoS através de mecanismos imperativos nas respectivas implementações [107].

Deve-se observar que muitas das questões relacionadas com o gerenciamento de QoS não podem ser implementadas isoladamente, dentro das fronteiras de um componente, devido as seguintes limitações:

- A provisão de QoS deve ser feita levando em consideração uma abordagem fim-a-fim. Ou seja, todos os elementos envolvidos numa interação podem ser afetados. A introdução de lógica de gerenciamento de QoS para dentro do componente pode afetar inclusive a própria reusabilidade do mesmo.
- Alguns aspectos relacionados com recursos específicos de baixo nível, como por exemplo, gerenciamento de *threads*, só podem ser suportados no contexto de um componente servidor de *threads*. Uma vez que desenvolvedores de componentes freqüentemente não têm conhecimento, a priori, sobre a forma como outros componentes devem operar, não parece correto deixar sob sua responsabilidade a tarefa de gerenciar a provisão de QoS.
- Certos mecanismos de garantia de QoS, tais como configurações de conexões não-multiplexadas entre componentes afetam a capacidade de interconexão. Como implementações

reusáveis de componentes não sabem antecipadamente como os componentes serão configurados numa composição, não é possível que uma implementação de componente trate todos os possíveis tipos de aprovisionamento de QoS isoladamente.

- Muitas políticas e mecanismos de suporte a QoS requerem a instalação de ferramentas que precisam de adaptações e configurações específicas. Entretanto, algumas políticas que requeiram, por exemplo, baixa latência de um canal e alta vazão, podem ser incompatíveis com as capacidades da plataforma. Assim, seria difícil prover suporte para QoS nos componentes sem conhecer os requisitos de todos os elementos envolvidos no contexto fim-a-fim.

Ou seja, trazer o gerenciamento de QoS para dentro do componente de forma prematura, requereria uma implementação para cada cenário. Este acoplamento afetaria e neutralizaria um beneficio básico do modelo de componentes: separação da funcionalidade do gerenciamento.

2.3 Requisitos para Sistemas de *Middleware* Adaptativos

As pesquisas relacionadas com arquiteturas e serviços de *middeware* têm evoluído bastante. O objetivo estabelecido consiste em suprir as necessidades que vão sendo impostas pelas novas aplicações. Assim, é importante que uma solução atual dê suporte à adaptação e inclusão de novas tecnologias; infelizmente isto ainda não ocorre na maioria das plataformas de *middeware*, uma vez que eles definem seus componentes como "caixas-pretas". Alguns *frameworks* como o OpenORB [20] e QuO [9] têm procurado dar suporte à adaptação em plataformas de *middleware* com abrangência mais genérica. Outros focam a adaptação no contexto de domínios específicos de aplicações, como os *frameworks* Syngo [116], TINA [14], RT-CORBA [117], TAO [118] e CM-TEL [19].

A abordagem do presente trabalho consiste em identificar requisitos gerais de projeto que dêem ao *middleware* características de generalidade e ao mesmo tempo, capacidades de adaptação para ser instanciado em ambientes específicos de suporte a aplicações multimídia. Os principais requisitos identificados são listados nas seções seguintes.

2.3.1 Conectividade em Plataformas Distribuídas Heterogêneas

Atualmente, existe uma variedade de tecnologias na área de telecomunicações que dão suporte à conectividade de vários tipos de dispositivos e plataformas, envolvendo desde supercomputadores e *desktops,* até dispositivos como PDAs e *Set-top-boxes*. Há também uma diversidade de tipos de aplicações; algumas delas, como as aplicações multimídia distribuídas precisam transmitir dados de fluxos de áudio e vídeo. Neste universo onde a conectividade envolve a integração de uma diversidade de elementos, pode-se constatar que muitos sistemas ainda não oferecem suporte adequado para realizar esta integração [97]. Neste contexto, este trabalho faz uma análise destes requisitos considerando como base os modelos para processamento distribuído aberto definidos nas arquiteturas de comunicação apresentadas nas propostas NMM[11] e RM-ODP[13].

O Modelo de Referência RM-ODP descreve uma arquitetura de comunicação que trata vários aspectos e conceitos relacionados com processamento distribuído. O *middleware* NMM – um *middleware* que incorpora muitos conceitos do modelo de referência ODP -, permite controlar e conectar dispositivos multimídia que utilizam diferentes tecnologias.

Para dar suporte às aplicações distribuídas, os conceitos de grupo e comunicação *multicast* são fundamentais. No entanto, tratar aspectos não-funcionais associados a parâmetros de QoS, *delay*, *jitter* e confiabilidade em canais *multicast*, onde poderão estar interconectados elementos de diferentes tecnologias, envolve muita complexidade; neste caso, o *middleware* deve tratar simultaneamente e de forma integrada, questões relacionadas com configuração, topologia de comunicação, semântica de comunicação, funcionalidade distribuída e orquestração (sincronização envolvendo relacionamentos entre diferentes fluxos).

Assim, considerando as análises apresentadas na literatura [7] [11] [13] [119], os seguintes requisitos foram identificados como base para dar suporte à conectividade:

- **Acesso aos componentes de forma transparente à distribuição**: para facilitar o desenvolvimento e a construção de uma aplicação distribuída, o *middleware* deve oferecer mecanismos para a aplicação acessar e controlar, de forma transparente à localização, os componentes da aplicação.
- **Interconexão de componentes de forma independente de distribuição:** antes de ativar a aplicação (transmissão do fluxo), é preciso realizar as conexões entre os componentes envolvidos no fluxo. Para prover flexibilidade, o *middleware* deve dar suporte ao estabelecimento de *bindings* flexíveis e configuráveis. A definição de *bindings* deve permitir a construção e gerenciamento de fluxos de forma transparente em relação aos conceitos de grupos,

tecnologias de comunicação, topologias e QoS. Como a maioria das soluções atuais tratam os componentes como "caixas-pretas", as definições de *bindings* nessas abordagens não permitem a realização de inspeções, nem tampouco alterações de comportamentos. Nestas abordagens, há dificuldades para introduzir mecanismos de monitoração e adaptação de parâmetros de QoS relacionados com, por exemplo, *jitter* e *delay*.

- **Transparência em relação à sincronização distribuída**: como os componentes de um fluxo distribuído estão potencialmente espalhados em diferentes computadores, o *middleware* precisa prover facilidades para a realização da sincronização distribuída. Neste caso, o processo de sincronização deve ser abstraído da aplicação, que deve se preocupar apenas com a especificação dos parâmetros de QoS associados.
- **Suporte à configuração e reconfiguração de fluxos em sistemas distribuídos.** Para prover flexibilidade, o *middleware* deve suportar a realização dinâmica de *bindings*, de modo a introduzir ou remover elementos associados a um fluxo.

Nos modelos ODP e NMM, as questões relacionadas com a realização de *bindings* são tratadas através de um componente denominado canal.

2.3.2 Abordagem Arquitetural

Sistemas multimídia distribuídos abertos devem ter capacidades para serem ajustados dinamicamente em função das características e estado da plataforma. Considerando que o nível de complexidade para tratar a adaptação no contexto das aplicações multimídia distribuídas é elevado, uma forma indicada para gerenciar estas questões é usar técnicas baseadas na arquitetura de *software*. Segundo Shaw [120], definir uma arquitetura de *software* consiste em descrever em alto-nível, os elementos arquiteturais envolvidos - "elementos visíveis externamente" -, os relacionamentos, as interações entre eles, os padrões de composição e as principais propriedades dos elementos da arquitetura.

Analisando os modelos definidos em CORBA [8], ODP [13] e a diversidade de requisitos e estilos arquiteturais [121] [122], foram identificadas e destacadas as seguintes características fundamentais para plataformas de suporte às aplicações multimídia distribuídas:

- **Generalidade:** o *middleware* deve identificar funcionalidades comuns para serem encapsuladas em componentes gerais, como por exemplo, componentes com funcionalidades de gerenciamento.
- **Extensibilidade**: O *middleware* deve permitir a definição e incorporação de novos componentes, assim como a introdução de novas funcionalidades.
- **Configurabilidade**: o *middleware* deve permitir que usuários controlem ou ajustem as funcionalidade e propriedades dos componentes de acordo com os propósitos da aplicação. Ou seja, o *middleware* deve permitir a seleção de protocolos adequados ao tipo da aplicação, ou por exemplo, em plataformas mais restritas como a de sistemas embutidos, permitir que se defina uma versão reduzida de funcionalidades. Aspectos não-funcionais também devem ser indicados.
- **Interoperabilidade**: o *middleware* deve permitir que elementos de uma plataforma distribuída sejam capazes de interagir com outros elementos na plataforma, independentemente das tecnologias ou linguagens que eles tenham sido implementados.
- **Evolução**: o *middleware* deve considerar que poderão ocorrer evoluções de forma que possa incorporar novas tecnologias como, por exemplo, diferentes tipos de mídia e novos algoritmos de compressão.
- **Suporte para componentes de granularidade fina e grossa**. O *middleware* deve prover suporte para gerenciar componentes de granularidade grossa, como por exemplo, sistemas e aplicações, bem como componentes com granularidade fina através dos quais são realizadas as construções dos sistemas e das aplicações.

2.3.3 Adaptação e QoS

Para dar suporte à adaptação dinâmica, o sistema deve permitir a realização de ajustes no seu comportamento em resposta, por exemplo, a mudanças dos requisitos da aplicação ou do ambiente. Existem várias situações em que adaptações poderão ocorrer, incluindo, entre outras: mudanças nos parâmetros de QoS da aplicação, mudanças nos parâmetros de QoS da rede, inclusão de novos componentes, detecção de desvios da QoS estabelecida e solicitação de adaptação originada por iniciativa da própria aplicação

Mudanças originadas por decisão do sistema (*middleware*) estão normalmente relacionadas a variações de QoS, ou à indisponibilidade temporária de recursos ou a adaptações pré-programadas. Em

particular, as questões de QoS são críticas no contexto dos sistemas multimídia distribuídos. Em razão disso, elas são vastamente discutidas na literatura; entretanto, o suporte ao gerenciamento desses aspectos ainda se constituem em desafios no contexto dos modelos de desenvolvimento baseados em componentes. A seguir, considerando algumas discussões levantadas na literatura [10] [123] [124], o texto destaca os seguintes requisitos para o gerenciamento de QoS:

- **Gerenciamento dinâmico de QoS**: aplicações devem ser capazes de especificar os requisitos de QoS desejados, assim como alterá-los dinamicamente. Cabe então ao *middleware* prover a QoS requisitada e suportar eventuais adaptações mediante mudanças de requisitos e disponibilidade de recursos.

- **Evolução dos requisitos de QoS:** como novos tipos de mídia e aplicações poderão surgir, o *middleware* deve ser capaz de incorporar novas características e algoritmos de gerenciamento de QoS.

- **Política de controle de QoS:** com o objetivo de prover características de simplicidade, o *middleware* deve ter condições para decidir automaticamente entre este ou aquele formato de mídia. No entanto, considerando o largo espaço que existe entre a visão do usuário e os parâmetros de QoS dos recursos de uma plataforma, muitas das decisões e escolhas precisam de diretrizes para condução do processo de mapeamento da visão do usuário, bem como da adaptação. Neste sentido, usuários devem ter a opção de especificar políticas de mapeamento, negociação, escolha de propriedades, monitoramento e adaptação. Por exemplo, deve-se permitir que a aplicação indique possíveis estados de degradação, bem como políticas a serem utilizadas pelo *middleware* para realizar a adaptação. Assim, para estabelecer que a QoS de um vídeo precisa ser monitorada, o desenvolvedor deve indicar as situações de degradação que requeiram ações por parte do *middleware*, bem como, a vinculação destas situações com possíveis políticas de adaptação, como por exemplo, a redução da taxa de quadros por segundo no processo de codificação do fluxo.

- **Análise e suporte automático da consistência da QoS**: o *middleware* deve analisar as capacidades dos recursos da plataforma e detectar possíveis inconsistências em relação aos requisitos da aplicação. Neste caso, se possível, o *middleware* pode inserir transcodificadores, provendo assim requisitos de consistência.

- **Suporte à adaptação reativa**: o *middleware* deve permitir a realização de ajustes originados por interação com usuários da aplicação que eventualmente desejarem selecionar determinadas propriedades no decorrer da execução da aplicação.
- **Suporte à adaptação pró-ativa**: o *middleware* pode tomar a iniciativa de mudar determinadas propriedades de componentes do sistema, ou mesmo trocar componentes quando perceber, por exemplo, que novos componentes que consomem menos recursos estão disponíveis.

2.3.4 Configuração e Reconfiguração Dinâmica

Para permitir a realização de ajustes provocados por mudanças na plataforma, ou nos requisitos da aplicação, um *middleware* precisa ser dinamicamente configurado (processo de instanciação), ou reconfigurado (adaptação por mudanças dinâmicas no estado do sistema).

Um exemplo muito comum em que o *middleware* precisa se ajustar está relacionado à contínua flutuação da carga no uso dos recursos da rede. Para dar suporte à reconfiguração, deve-se preservar o nível de consistência envolvendo o estado interno do *middleware*, as necessidades impostas pelas possíveis mudanças e as disponibilidades da plataforma. O grande desafio é realizar a transição mantendo o estado do sistema consistente. Os mecanismos de reconfiguração devem ser genéricos, de modo a admitir um conjunto extenso de algoritmos de adaptação. A execução de uma operação de reconfiguração requer cuidados especiais no tratamento de interações durante a fase de transição de estado. A concretização de uma transação pode envolver atividades como a alocação e liberação de recursos, a troca de formatos e propriedades dos componentes, ou até mesmo a substituição de algoritmos e componentes.

Deve-se destacar portanto que configuração e reconfiguração dinâmicas se constituem em pontos importantes para serem tratados pelas atuais tecnologias de *middleware.* Assim, segundo as análises realizadas por McKinley [10] e Blair [125], os seguintes requisitos devem ser observados em um projeto de *middleware* adaptativo para sistemas multimídia distribuídos:

- **Permitir acesso a funcionalidades do *middleware*, sistema operacional e da rede:** o princípio consiste em usar os conceitos de reflexividade e reificação apresentados por Szypersky [27] para definir operações de introspecção e ajuste de valores de propriedades, de forma a realizar o gerenciamento de propriedades dinâmicas do sistema.

- **Suportar o desenvolvimento de aplicações usando um modelo de configuração neutro em relação a tecnologias**: o princípio consiste que a configuração dos componentes da aplicação e do *middleware* não seja vinculada a uma linguagem específica; deve-se no entanto manter a aderência a um modelo de desenvolvimento baseado em componentes.
- **Acomodar sistemas legados**: o *middleware* deve prover mecanismos que permitam a integração e utilização de sistemas legados, como sistemas operacionais, protocolos, serviços e ORBS, como por exemplo CORBA e COM.

2.4 Considerações Finais

Neste capítulo, foram abordadas as principais características das aplicações e sistemas multimídia distribuídos. O texto destacou que as tecnologias de *middleware* constituem-se em alternativas interessantes para tratar a complexidade do processo de desenvolvimento desses sistemas. No entanto, algumas limitações foram identificadas em relação às arquiteturas das plataformas de *middleware* atuais. Com o objetivo de superar estas limitações, foram identificados alguns requisitos que devem ser observados nos projetos de *middleware*, de modo a prover suporte à adaptação nos atuais e futuros sistemas. Estes requisitos servem de base também para realizar, no próximo capítulo, uma análise envolvendo o estado da arte, onde alguns trabalhos relacionados são discutidos.

Capítulo 3

Estado da Arte

A definição de uma arquitetura para desenvolvimento e configuração de sistemas multimídia distribuídos envolve uma diversidade de requisitos, conforme foi abordado no capítulo anterior. Vários *frameworks* e plataformas de *middleware* foram propostos nos últimos anos com o objetivo de dar suporte a estes requisitos. Dentre estes, o presente capítulo discute alguns devido à relevância dos mesmos em relação aos requisitos identificados no capítulo anterior.

Considerando o fato do Cosmos ter sido implementado no contexto de um protótipo de um *middleware* adaptativo para sistemas de Televisão Digital Interativa, o AdapTV, o capítulo analisa também algumas questões relacionadas com plataformas de *middleware* para sistemas de televisão digital interativa.

3.1 Abordagens Arquiteturais e Conectividade

Nesta seção são discutidas algumas arquiteturas de *middleware* que tratam questões gerais relacionadas a desenvolvimento e suporte de sistemas multimídia distribuídos. Estes trabalhos são importantes à medida que foram utilizados como base para a análise de requisitos dos modelos arquitetural e de interconexão de componentes definidos para o *framework* Cosmos.

3.1.1 O Modelo PREMO

O *framework* PREMO (*Presentation Environment for Multimedia Objects)* [7] [128], proposto pela ISO, leva em consideração um extenso conjunto de requisitos para aplicações multimídia distribuídas. O PREMO define um modelo de objetos e de sincronização, assim como um conjunto de serviços multimídia. Os principais elementos da arquitetura de PREMO são as abstrações para dispositivos (*VirtualDevice*s), portas (*port*) e conexões virtuais (*Virtual Connections*). Estas entidades

estão associadas às noções de interfaces de fluxo e aos conceitos de *bindings* do modelo RM-ODP [13], suportando conexões *ponto-ponto* e *ponto-multiponto*. Esta seção discute brevemente estes elementos do PREMO.

Um recurso virtual encapsula uma unidade de processamento de mídia, podendo ser uma entidade de *software*, como um CODEC, ou uma entidade de *hardware*, como uma placa de captura de áudio ou vídeo. A idéia é que a abstração trate diferentes tipos de dispositivos e tecnologias, como por exemplo, uma placa PCI e um sintonizador de TV de forma uniforme.

Para realizar uma interconexão entre recursos virtuais, PREMO introduz o conceito de portas. Uma porta tem um papel similar ao de uma interface de fluxo do RM-ODP. Uma porta pode ter um ou mais formatos de mídia, podendo ser consultados e configurados. Para realização de controle e sincronização, recursos virtuais têm associado uma interface de controle de fluxo.

Uma conexão virtual é um objeto que abstrai a questão de gerenciamento e transferência de dados de mídia entre recursos virtuais. A conexão virtual na verdade não realiza a transferência. Ela é responsável por separar a conexão em diferentes elementos, permitindo assim, a realização de controle e negociações sobre as propriedades do fluxo e o suporte adequado em termos de quais tecnologias utilizar para realizar a transferência.

O estabelecimento de uma conexão no PREMO envolve a definição de um tipo (local ou remota), a negociação de formatos do fluxo, a QoS e as capacidades dos elementos envolvidos.

Uma limitação do modelo para conexão virtual definido no PREMO é não permitir que a aplicação se envolva na definição de propriedades do processamento e transporte do fluxo, o que inviabiliza a realização de adaptação. Cabe ao programador o papel restrito de apenas definir os dispositivos e conectá-los. PREMO não permite acessar propriedades que ficam escondidas dentro do adaptador de conexão.

Os conceitos de propriedades e portas no PREMO são usados, respectivamente, com a finalidade de descrever as características dos recursos virtuais, para fins de configuração, e de estabelecer elementos de interação envolvendo fluxos. No PREMO, as portas aparecem inclusas nos Recursos, – na verdade, em dispositivos virtuais -, que na arquitetura correspondem a classes concretas associadas a recursos virtuais (classes abstratas). O tratamento da variabilidade de opções de dispositivos no PREMO se concentra na implementação do recurso virtual através do uso da técnica de herança. Assim, pode-se considerar PREMO como um *framework* do tipo caixa-branca.

O *framework* PREMO possui um grande nível de complexidade, o que inviabiliza a sua aplicação no contexto de desenvolvimento de sistemas multimídia atuais, como por exemplo, Sistemas de Televisão Digital Interativa, onde dispositivos possuem normalmente limitações de *hardware*. Talvez, devido a esta complexidade, é incomum encontrar referências para sua implementação na literatura. Cabe no entanto destacar que os conceitos do PREMO influenciaram outras propostas como o *framework A/V Streams* [8] e a própria definição do Cosmos.

3.1.2 O *Framework A/V Streams* da OMG

O *framework A/V Streams* da OMG [8] é um padrão aberto aderente à arquitetura CORBA. A especificação define componentes e serviços com uma abordagem arquitetural para implementação e controle de aplicações envolvendo fluxos multimídia. *A/V Streams* trata configurações de fluxo ponto-ponto e ponto-multiponto. Conforme esta especificação, todas as operações de controle são realizadas utilizando o ORB. No entanto, os dados de fluxo de mídia são transportados por fora do ORB, através de protocolos adequados (por exemplo, TCP, UDP, RTP/UDP, etc.).

A representação, gerenciamento e configuração de recursos ocorre através dos conceitos de *flow endpoints* - pontos terminais de fluxo -, podendo ser especializados para se comportarem como produtores de fluxo (*Flow-Producer*), ou consumidores de fluxo (*Flow-Consumer*). Uma configuração pode realizar conexões (*flow connection*) envolvendo, por exemplo, um produtor e um ou mais consumidores. Uma interface denominada *FlowConnection* abstrai o conceito de conexão de fluxo. Com relação ao gerenciamento de QoS, fica a cargo da aplicação mapear os parâmetros de QoS, tais como taxa de transmissão, como por exemplo, *framerate* em parâmetros do tipo *byterate*.

A arquitetura do *framework* A/V *Streams* teve como base o modelo PREMO, cobrindo assim vários aspectos dos sistemas multimídia distribuídos. Adicionalmente, devido ao fato do A/V *Streams* herdar as características do modelo arquitetural do CORBA, ele incorpora as várias facilidades do CORBA além do suporte a processamento distribuído aberto. O *framework* A/V *Streams* também usa o conceito de propriedades para dar suporte à configuração e reconfiguração. Como a estrutura da arquitetura do A/V *Streams* é semelhante à definida no PREMO, as limitações do PREMO também ocorrem no contexto do A/V *Streams*. Outra observação a ser destacada em relação ao PREMO e ao A/V *Streams* é que nestas abordagens a reconfiguração se restringe ao modelo de atuação reativa, não suportando adaptação pró-ativa.

3.1.3 O *middleware* NMM

O *middleware* NMM (*Network-Integrated Multimedia Middleware*) [11] [12] oferece uma arquitetura para a construção de aplicações multimídia distribuídas em ambientes heterogêneos. Como elemento principal desta arquitetura destaca-se o conceito de nó que encapsula processamento e funcionalidades. Os nós se conectam com outros nós através dos *jacks*. Os *jacks* são responsáveis por transportar os dados dos *buffers* de um nó para o outro, representando a entrada e saída de um nó. O NMM explora o conceito de portas através do uso de *jacks,* possibilitando conexões ponto-ponto e ponto-multiponto de forma transparente para o componente que está enviando os dados.

Construir uma aplicação no NMM consiste em definir um grafo envolvendo os elementos arquiteturais. Este grafo deve ser criado de acordo com os recursos disponíveis no sistema. Assim, a arquitetura do NMM trata os aspectos de reconfiguração dinâmica e QoS focando a abordagem nos grafos de fluxos ativos. Para criar grafos envolvendo conexões entre elementos da arquitetura, o NMM define um *framework* para realização de *binds*, criando canais de comunicação (elementos responsáveis por realizar as interações). O tratamento de QoS e adaptação dinâmica são limitados, aparecendo na arquitetura no escopo da sincronização, em canais paralelos [129]. Canais paralelos são utilizados para definição de diferentes QoS onde cada canal pode usar uma tecnologia diferente. O modelo provê abstrações para que aplicações, ou o próprio *middleware* possam realizar reconfigurações dinâmicas. Entretanto, Lohse [12] afirma que o suporte para gerenciamento de QoS no NMM ainda se encontra em desenvolvimento e devido a isso não fornece maiores detalhes.

As propostas NMM, PREMO e A/V *Streams* representam aplicações multimídia como grafos, onde os elementos são relacionados num estilo arquitetural do tipo *pipe-filter*. O conceito de canal nestas abordagens é originado da proposta ODP [13] e a visão de reconfiguração de QoS no NMM se dá através do uso de canais paralelos. Entretanto, este mecanismo usado no NMM no processo de adaptação é bastante complexo.

3.1.4 Modelo de Componentes CM-tel

CM-tel foi definido por Guimarães [19] com o objetivo de dar suporte ao desenvolvimento e execução de aplicações telemáticas e ubíquas. CM-tel propõe um modelo neutro em termos de tecnologia, sendo especificado por meio da linguagem UML (*Unified Modeling Language*). Os

componentes do CM-tel podem ser executados tanto em plataformas tradicionais quanto em dispositivos com limitações computacionais, tais como dispositivos móveis. CM-tel apresenta os três tipos de interfaces definidas no modelo RM-ODP, ou seja, as interfaces operacional, de sinal e de fluxo contínuo. Para o contexto das aplicações telemáticas, as quais geralmente envolvem requisitos de aplicações multimídia, as interfaces de fluxo contínuo são fundamentais.

A arquitetura de gerenciamento definida no contêiner CM-tel integra componentes e agentes móveis em um único ambiente computacional. Esta integração permite que aplicações implementem suas funcionalidades combinando componentes e agentes móveis. O trabalho também propõe uma arquitetura para plataformas de desenvolvimento de *software* aderentes ao modelo CM-tel. Esta arquitetura utiliza XSLT (*XML Stylesheet Language Transformation*) para transformação de modelos e geração de código. Esta proposta foi validada experimentalmente através do desenvolvimento de uma plataforma baseada na tecnologia CORBA.

3.2 Adaptação e QoS

Nesta seção são discutidas algumas abordagens empregadas atualmente para dar suporte à adaptação. Na discussão são considerados aspectos relacionados a gerenciamento de recursos, QoS e computação reflexiva.

3.2.1 OpenORB

OpenORB [20] apresenta uma arquitetura de *middleware* reflexivo. A arquitetura foi definida e estruturada usando o conceito de *framewok* de componentes [100]. O *framework* inclui modelos para representação e gerenciamento de recursos e modelos para gerenciamento de tarefas.

No OpenORB, os metamodelos que descrevem a interface e a arquitetura (aspectos arquiteturais) são tratados separadamente dos metamodelos de recursos e de interceptação (aspectos comportamentais). Os metamodelos comportamentais são utilizados para observar e controlar as atividades dos componentes (por exemplo, mensagens e *threads*) para efeito de gerenciamento de recursos [49] [100].

O modelo de recursos oferece vários níveis de abstração com o objetivo de representar os subsistemas de recursos, ao passo que o modelo de tarefas tem como função prover mecanismos de

alto-nível para realização de consultas, análises e projetos de gerenciadores de recursos de sistema. O modelo de tarefas permite realizar gerenciamento de recursos envolvendo interações com nível de granularidade grossa e fina.

O desenvolvimento de plataformas de *middleware* reflexivo, conforme esta arquitetura, provê potencialidades para suportar mecanismos de configuração e reconfiguração dinâmicas. A reconfiguração corresponde à realização de *bindings* através da navegação nos grafos de metadados que mantêm a descrição da arquitetura do sistema.

No que se relaciona aos mecanismos definidos para suportar adaptação, o OpenORB utiliza metacomponentes que descrevem os elementos estruturais da arquitetura. Porém, na análise realizada, observou-se que os *frameworks* arquiteturais para gerenciamento de recursos e tarefas no OpenORB [100] [130] apresentam abordagens com níveis elevados de complexidade. Estes *frameworks* definem hierarquias para recursos e tarefas com uma visão em que os componentes do *middleware* fornecem funcionalidades com níveis de granularidade similares a de um sistema operacional convencional, envolvendo operações de baixo nível para escalonamento de tarefas e para alocação de recursos.

Outro aspecto em relação ao OpenORB diz respeito à estratégia adotada para suportar diferentes tipos de conexões (*bindings*). De acordo com Guimarães [19], para adicionar novas funções para cada tipo de conexão que a aplicação venha a requerer, o OpenORB requer que a API básica seja estendida pelo desenvolvedor da aplicação. Esta estratégia requer do desenvolvedor o fornecimento de todo o código para a realização da conexão.

3.2.2 TAO

O projeto TAO [131] teve como preocupação prover suporte em tempo-real para ORBs baseados no CORBA com objetivo de tratar requisitos de QoS nestas plataformas. O TAO foi implementado sobre o ACE [16] de acordo com as especificações da OMG para POA (*Portable Object Adapter*), fornecendo uma infra-estrutura eficiente em tempo de execução para comunicações unidirecionais ou bidirecionais, de forma síncrona ou assíncrona entre objetos distribuídos de aplicações. O ORB gerencia conexões de transporte, envio de requisições cliente e retorno das respectivas respostas.

TAO usa o padrão de projeto *Strategy* [127] para separar os diferentes aspectos do ORB. Um arquivo de configuração é usado para especificar as estratégias usadas pelo ORB para implementar aspectos como concorrência, demultiplexação de requisições, escalonamento e gerenciamento de

conexões. Na iniciação do sistema, o arquivo de configuração é processado e as estratégias selecionadas carregadas.

TAO foi definido no contexto de aplicações de tempo real crítico. Uma vez configuradas, as estratégias permanecem até o fim da execução, ou seja, praticamente esta abordagem não suporta reconfiguração dinâmica.

O TAO provê suporte para implementação de ligações explícitas (*explicit bindings*) definidas pelo RT-CORBA. As conexões de *cache* do TAO são estendidas para garantir propriedades de QoS das conexões, as quais incluem prioridade de banda e atributos privados.

Para dar algum nível de suporte à adaptação dinâmica, foi definido o *middleware dynamicTAO* [132], uma extensão ao TAO. *DynamicTAO* exporta uma interface para carga e descarga de módulos no ORB em tempo de execução e para inspeção do estado do ORB. A arquitetura definida também pode ser usada para reconfiguração dinâmica de *servants*.

Cada processo em execução contém uma instância de um componente *ComponenteConfigurator*, denominada Configurador de domínio. Esta instância mantém as referências para as instâncias do ORB e dos *servants* daquele processo. Em adição, cada instância do ORB contém uma subclasse especializada de *ComponentConfigurator*, chamada *TAOConfigurator*.

O *TAOConfigurator* define "ganchos" para as estratégias suportadas pelo ORB *dynamicTAO*. Um *broker* de comunicação (*NetworkBroker*) implementa um protocolo baseado em TCP para conexão de entidades remotas com o objetivo de realizar inspeções e mudanças no configurador *dynamicTAO*, carregando, por exemplo, novas estratégias. *Servants* locais e clientes CORBA remotos também podem acessar objetos *Configurators* através de interfaces CORBA.

3.2.3 QuO

O QuO (*QoS for CORBA Objects*) [9] foi projetado para dar suporte a QoS na camada de objetos CORBA. O objetivo é permitir o ajuste do comportamento do sistema de acordo com as variações de QoS ocorridas durante a execução. Os principais elementos de QuO são:

- ***Contracts*:** objetos que representam regiões associadas a requisitos de QoS da aplicação e as ações relacionadas com a adaptação;

- ***Delegate***: objetos que se comportam como representantes dos clientes (*proxy*) oferecendo a mesma interface, porém acrescentando código para verificar o estado da QoS da região, podendo introduzir adaptação.
- ***System Condition***: objetos que provêem a interface para os recursos, mecanismos e serviços do sistema.

QuO define uma linguagem para descrição dos requisitos; na verdade ele estende a funcionalidade da IDL (*Interface Definition Language*) com um grupo de descrições separadas para tratar aspectos de requisitos de QoS (QDL). Por ser uma extensão da IDL, uma especificação de QoS em QDL pode ser mapeada para várias linguagens. A linguagem QDL oferece facilidades para especificar contratos (CDL - *Contract Description Language*), recursos (RDL - *Resource Description Language*) e adaptação no comportamento de objetos *delegates* (SDL- *Structure Description Language*).

Recursos são acessados por elementos *system conditions*; dessa forma, o suporte para manipulação e consulta ao estado de um recurso fica restrito a determinadas circunstâncias, uma vez que no *framework* QuO a questão da QoS é tratada através de mecanismos associados a interações originadas em objetos clientes, as quais são interceptadas por elementos *delegate*.

QuO permite especificar faixas de aceitação e associar ações que devem ser tomadas caso os requisitos de QoS não sejam atendidos. Quando o sistema passa para uma região de menor QoS que a região atual, depois de um determinado tempo (definido na especificação), ele tenta subir para uma região de melhor QoS. Uma limitação de QuO nesse processo consiste em tentar melhorar a QoS sem verificar a consistência da ação. Ou seja, o processo verifica apenas se o componente servidor pode oferecer um nível melhor de QoS e, sem analisar se a plataforma do cliente suporta esta QoS, o suporte dá início ao processo de adaptação [83] [84]. Assim, uma aplicação que estiver na região de maior QoS suportada pela plataforma ficará tentando ir para uma região de melhor QoS, embora não exista tal região.

3.2.4 Agilos

Agilos [133] define um *framework* no nível do *middleware* que provê suporte para adaptação de QoS. O *framework* foca a adaptação na visão da aplicação. O modelo definido pelo *framework* consiste de três camadas:

- Primeira camada: inclui *observers* e *adaptors*, que são elementos independentes da aplicação. *Observers* são responsáveis por monitorar os recursos disponíveis e inspecionar os valores de QoS específicos da aplicação. Cada tipo de recurso tem um tipo específico de *observer*. *Adaptors* são também específicos para cada tipo de recurso e são responsáveis por sinalizar quando houver variações na QoS indicadas por *observers*.
- Segunda camada: inclui objetos *configurators* e *qualProbes*. *Configurators* são responsáveis pela coordenação e controle das ações a serem executadas, dependendo dos sinais gerados pelos *adaptors* e dos requisitos de QoS estabelecidos pela aplicação. *Configurators* desempenham um papel fundamental inspirado na teoria de controle baseada na Lógica *Fuzzy*. *QualProbes* são regras interpretadas pela máquina de inferência, gerando ações que controlam a aplicação. Uma das principais finalidades do *QualProbes* é permitir que as aplicações especifiquem os requisitos de QoS.
- Terceira camada: inclui objetos *gateway* e *negotiators*. Existem vários *negotiators* e um *gateway* centralizado. O modelo permite um cliente se conectar em diferentes servidores de acordo com as necessidades de adaptação do cliente. *Negotiators* podem estar tanto em clientes quanto em servidores. Um *negotiator* num cliente interage com o *Configurator* e com o *gateway* para realizar uma mudança de comportamento. *Negotiators* em servidores somente interagem com observadores e com o *gateway*. Como *configurators* são passivos no lado do servidor, *negotiators* nos servidores não precisam interagir com eles. *Configurators* nos servidores são passivos porque em Agilos as decisões e estratégias de adaptação são definidas apenas pelos clientes. O *gateway* é responsável por manter as informações consistentes nos clientes e servidores e por escolher o servidor mais adequado para atender a requisição do cliente. Para isso, o *gateway* considera as informações enviadas pelo *negotiator* do cliente.

Agilos define um *middleware* baseado na Teoria de Controle *Fuzzy* para decisão de que ações tomar. Neste caso, o gerenciador funciona como um "controlador" que é realimentado com informações sobre os comportamentos dos recursos que são monitorados por observadores específicos. No entanto, no Agilos não há uma representação explícita para manipular recursos; o foco do Agilos está concentrado nos aspectos da adaptação no nível da aplicação, onde são definidas as semânticas das estratégias de adaptação. Desta forma, Agilos não dá suporte à configuração, nem reconfiguração de recursos.

3.3 Configuração e Reconfiguração Dinâmicas

Várias abordagens têm sido exploradas para dar suporte à programação de configuração e reconfiguração. Em geral, uma configuração envolve descrições de parâmetros para implantação (*deployment*) e instanciação de componentes, assim como para configuração e reconfiguação dinâmica do *middleware* e da aplicação. As abordagens que tratam a configuração com requisitos de adaptação dinâmica geralmente utilizam os princípios da computação reflexiva, onde normalmente são explorados conceitos de metadados e metacomponentes. Nestes casos, o gerenciamento da adaptação requer a manutenção atualizada das informações representadas nos metadados. Assim, a programação da adaptação pode ser tratada com diferentes visões, descritas através da definição de metacomponentes, ou da especificação através de linguagens para metaprogramação, conforme discutido na introdução da presente tese. As seções seguintes analisam algumas soluções usadas em propostas relacionadas.

3.3.1 Gerenciamento de Metadados e Reflexão no OpenORB

Uma proposta interessante e flexível para gerenciamento de metadados foi apresentada no contexto do projeto OpenORB [134]. Nesta proposta, os conceitos de metadados e reflexão foram definidos em um metaespaço estruturado, mas não restrito, em um conjunto de quatro metamodelos ortogonais: interface, arquitetura, interceptação e recursos.

O metamodelo para descrição de interfaces permite a introspecção nas interfaces fornecidas por um dado componente. O metamodelo que descreve a arquitetura representa componentes compostos através de grafos de componentes, onde conexões (*bindings*) são mapeadas em arestas. O metamodelo de interceptação é responsável por introduzir novos comportamentos para uma interface de componente. Por último, o metamodelo de recursos preocupa-se com os recursos e gerentes de recursos da plataforma. Esta abordagem é interessante no sentido de que permite a reusabilidade de configurações que estejam armazenadas em um determinado repositório de metadados.

Para programar a adaptação no OpenORB foram definidas duas linguagens [100]: uma linguagem de descrição arquitetural, denominada Xelha, e uma linguagem de descrição da configuração dos recursos, denominada RCDL (*Resource Configuration Description Language*).

A primeira consiste em uma linguagem de alto-nível usada para programar atividades de gerenciamento de QoS, enquanto que a segunda fornece suporte de baixo-nível para a especificação de

recursos e de políticas de gerenciamento. Dessa forma, os requisitos descritos em Xelha devem ser mapeados em RCDL.

O modelo de recursos tem como elementos mais importantes os recursos abstratos, a fábrica de recursos e os gerentes de recursos.

Os recursos abstratos representam explicitamente os recursos de sistema. Em adição, existem vários níveis de abstração, onde recursos de mais alto nível são construídos a partir dos de nível mais baixo. Os gerentes de recursos podem ser especializados para gerenciar, por exemplo, a carga do processamento associada aos recursos, assim como às *threads* ou processadores virtuais. Componentes fábricas de recursos têm a responsabilidade de criar recursos abstratos.

O modelo utiliza uma hierarquia de classes envolvendo os recursos no *framework*. A hierarquia de classes pode ser utilizada para uma instanciação em particular. A interface provê uma maneira transversal de se navegar na hierarquia de maneira recursiva; assim, pode-se aplicar operações em recursos de mais baixo nível ou em recursos de alto nível.

A hierarquia de fábricas provê uma interface que expõe operações para criar recursos de baixo nível, tendo também operações para fazer associações entre esses recursos e seus gerenciadores. Um recurso somente é criado se suas dependências estiverem disponíveis.

Processos de introspecção e reconfiguração utilizam essa hierarquia para localizar um recurso específico; para isso, eles navegam nos respectivos componentes utilizando a interface até chegar ao recurso procurado, onde é feita a configuração do recurso.

Xelha engloba definições de estruturas arquiteturais e de propriedades de QoS. Em comum com a maioria das ADLs, Xelha é usada para descrever os componentes, conectores, as interfaces e as definições estruturais envolvendo as tarefas que compõem um grafo estrutural. A linguagem também suporta a especificação de arquiteturas dinâmicas, ou seja, arquiteturas que podem ser ajustadas dinamicamente. Assim, pode-se introduzir o conceito de reuso de definições de configurações de componentes.

A RCDL é usada para especificar o gerenciamento de recursos. Esta linguagem adota o princípio da Programação Orientada a Aspectos; assim, uma definição Xelha é mapeada em diferentes aspectos. Seguindo a orientação de separação de interesses, uma especificação de serviços é mapeada para uma linguagem SDL (*Structure Description Language*), a qual fornece informações sobre o nível de QoS, a *task* e as classes de objetos que estão associadas com o serviço. O grafo de tarefas em Xelha é traduzido para as linguagens TSDL (*Task Switch Description Language*) e TDL (*Task Description*

Language). Propriedades de QoS das *tasks* são especializadas através de descrições RDL (*Resource Description Language*), que definem requisitos específicos da instância de acordo com as propriedades da plataforma. Por fim, a estrutura de gerenciamento de QoS é descrita na linguagem QMGDL (*QoS Management Graph Description Language*).

Além dos problemas anteriormente citados na Seção 3.2.1 com relação a aspectos de baixo-nível relacionados com as funcionalidades de sistema operacional, a diversidade de modelos associados ao OpenORB tornam a programação no *framework* muito complexa.

3.3.2 Gerenciamento de Configuração no CM-tel

O modelo de componentes CM-tel trata a configuração de uma aplicação em dois níveis de abstração:

- Primeiro nível: consiste de um modelo neutro em termos de tecnologias (independente de plataformas, linguagens de programação, sistemas operacionais, protocolos de rede, etc). Neste nível, o componente é especificado em UML ou XML, podendo ser mapeado para diferentes tecnologias.
- Segundo nível: requer o uso de plataformas de suporte ao modelo. Estas plataformas são construídas de acordo com o modelo de implementação física através de transformações de documentos XML. Entretanto, para que isto ocorra, é necessária a definição de mapeamentos do modelo para cada tecnologia. Estes mapeamentos acrescentam certas particularidades da tecnologia. Guimarães [19] descreve uma experiência de implementação de um protótipo da plataforma usando a tecnologia CORBA e a linguagem de programação Java.

No mapeamento do modelo CM-tel, as especificidades relacionadas às tecnologias ficam restritas aos *contêiners* e aos descritores de distribuição. *Contêiners* são elementos que fornecem o suporte necessário em tempo de execução aos componentes, bem como à infra-estrutura necessária para que estes componentes interajam com outros. Os descritores de distribuição especificam as características e os atributos necessários para personalizar o *contêiner* e seus componentes durante a sua instalação.

No caso dos *contêiners*, as especificidades são geradas automaticamente pela plataforma, enquanto que para descritores de distribuição alguns elementos novos precisam ser definidos, necessitando ser especificados de forma declarativa em XML.

A QoS para os fluxos de mídia é descrita inicialmente no nível da aplicação. Estas informações são usadas na configuração inicial do *contêiner*, durante a sua instalação e no monitoramento e controle de QoS. O CM-tel provê facilidades para que a aplicação ou o usuário escolha certos parâmetros de QoS, ou selecione os mesmos com base, por exemplo, no desempenho da rede. O monitoramento e controle de QoS consiste em monitorar os valores dos parâmetros de QoS da rede e corrigir discrepâncias.

Os parâmetros de instalação presentes no descritor de distribuição são armazenados como propriedades de configuração do *contêiner*. Por meio destas propriedades e após a instalação do *contêiner*, os parâmetros de instalação iniciais podem ser inspecionados e alterados.

O gerenciamento e controle de QoS é realizado empregando agentes móveis utilizando duas formas de adaptação dinâmica, onde as propriedades de QoS são mantidas e gerenciadas pelo *contêiner* e por agentes móveis. No exemplo explorado por Guimarães [19], o monitor foi implementado por código móvel. Este monitor utiliza um agente móvel que monitora certos parâmetros de QoS. Com base em um conjunto de regras de decisão, este agente atua no sentido de aprimorar a qualidade do vídeo estabelecida entre um componente produtor e um componente apresentador de vídeo, em função do desempenho da rede.

Um Monitor de QoS recebe como entrada um conjunto de fluxos de vídeo para fins de monitoramento e controle de QoS e obtém os parâmetros que foram utilizados na instalação do *contêiner*, como por exemplo, a taxa de bits (banda) ideal para o fluxo. Para cada fluxo, a tarefa de monitoramento e controle de QoS é delegada para uma instância do Agente de QoS. Este agente reporta-se periodicamente ao monitor de QoS que, de posse das informações obtidas pode notificar o usuário, renegociar a qualidade de serviço, ou ainda restabelecer o fluxo com novos parâmetros de QoS (por exemplo, tamanho da janela e taxa de quadros).

3.4 Sistemas para Televisão Digital Interativa

Como o Cosmos foi instanciado no contexto de um *middleware* para sistemas de Televisão Digital Interativa, o *middleware* AdapTV [31] [76] [77], esta seção apresenta uma breve discussão sobre as tecnologias adotadas pelos atuais modelos e padrões para TVDI.

De acordo com a análise feita por Leite [88], todas as propostas de sistemas de Televisão Digital especificam plataformas de *middleware* sobre as quais as aplicações de TV Interativa podem ser

executadas. Na verdade, o que estas propostas definem não corresponde exatamente ao conceito de *middleware* adaptativo explorado na presente tese. As propostas apresentam descrições de camadas com definições de APIs que têm como objetivo facilitar o uso das tecnologias associadas. As APIs definem operações e serviços para:

- gerência de aplicações e recursos do *middleware*;
- comunicação entre aplicações e processos;
- processamento (decodificação e exibição) de mídias;
- sincronismo;
- seleção, acesso e interpretação de fluxos elementares e informações de serviço;
- rastreamento de eventos e elementos áudio visuais de interface com o usuário;
- armazenamento, localização e recuperação de dados locais; e
- controle do ciclo de vida das aplicações.

Dentre as principais propostas de *middleware* para TVDI, podem ser citados: o MHP [135] para o sistema DVB; o DASE [136] e o ACAP [137] para o sistema ATSC; o ARIB-STD 24 [138] para o sistema ISDB e o OCAP [139], para o sistema de TV a cabo norte-americano. Estas soluções utilizam a tecnologia Java da Sun Microsystems, explorando as seguintes APIs:

- JavaTV [140], que viabiliza o desenvolvimento de aplicações interativas portáteis, de forma independente da tecnologia do *hardware* e da rede de difusão utilizados. Ela estende o pacote J2ME (Java *2 Platform, Micro Edition*) e adiciona funcionalidades específicas para o contexto de TVD;
- JMF [141], que define o comportamento e a interação de objetos de mídia contínua. A JMF é utilizada para capturar, processar e apresentar alguns tipos de mídia contínuas;
- DAVIC [142], que define requisitos de sistemas audiovisuais para prover interoperabilidade fim-a-fim; e
- HAVi [143], que estende o pacote gráfico padrão do Java (AWT) e adiciona funcionalidades para prover suporte gráfico para controle remoto, específico para TVD, entre outros.

Como forma de prover uma padronização única para *middleware* de TV digital, está sendo desenvolvido o padrão GEM (*Globally Executable* MHP) [144], contendo um conjunto de funcionalidades básicas (por exemplo, a compatibilidade com a tecnologia Java e utilização de APIs comuns) a todos os outros padrões de *middleware*. Nota-se, portanto, que é desejável que qualquer

middleware a ser desenvolvido para TV digital seja compatível com o modelo GEM, de forma a possibilitar a execução de um maior número de aplicações disponíveis e a serem desenvolvidas.

Apesar da grande quantidade de serviços e funcionalidades oferecidas pelas plataformas de *middleware* definidas pelos principais Sistemas de Televisão Digital existentes, inclusive o próprio GEM, a grande maioria deles não oferece os serviços inerentes aos sistemas de *middleware* de nova geração, tais como sensibilidade ao contexto, reconfigurabilidade, adaptabilidade, reflexibilidade, ubiqüidade, mobilidade etc., ou mesmo alguns serviços comumente encontrados nas plataformas de *middleware* de propósito geral, como provisão de QoS e tolerância a falhas [88]. Por este motivo, há espaço para novas propostas de *middleware* no domínio de sistemas de TVDI. Nesse sentido, o presente trabalho decidiu explorar este espaço no contexto do *middleware* AdapTV [31]. Este *middleware* foi desenvolvido como prova de conceito para o *framework* Cosmos.

3.5 Considerações Finais

Este capítulo apresentou algumas análises sobre o estado da arte envolvendo os principais trabalhos relacionados a plataformas de *middeware* para sistemas multimídia distribuídos. Apesar do texto apontar algumas limitações destas soluções relacionadas à totalidade dos requisitos enumerados no Capítulo 2 e à complexidade, vale destacar que as idéias discutidas são interessantes, sendo a maioria delas consideradas na proposta do *framework* Cosmos. As principais diferenças desses trabalhos em relação ao Cosmos estão relacionadas à abordagem para construção de aplicações. Segundo o modelo definido no Cosmos (apresentado no Capítulo 4), idealmente uma aplicação é construída utilizando componentes pré-existentes, usando por exemplo, modelos de distribuição como o do sistema SOS definido por Elias [147]. Esta visão assume que o processo envolve uma diversidade de elementos, e que estes elementos são desenvolvidos usando modelos e metodologias diferentes. Nesta linha, o Capítulo 4 apresenta o *framework* Cosmos, ressaltando que o mesmo foi definido para tratar estas questões buscando dar suporte aos requisitos de adaptação estabelecidos no capítulo anterior.

Capítulo 4

O *Framework* Cosmos

Os capítulos anteriores apresentaram os principais requisitos e o estado da arte relacionados a plataformas de *middleware* para desenvolvimento e execução de aplicações multimídia distribuídas. Vários aspectos foram considerados e destacados devido ao nível de importância dos mesmos para a formulação da proposta do *framework* Cosmos.

A análise mostrou que as soluções atuais, mesmo utilizando os conceitos de *middleware,* ainda são limitadas em relação a alguns aspectos, como por exemplo:

- Implementações restritivas para a abordagem cliente-servidor;
- Tratamento de QoS localizado;
- Nível de complexidade para tratar a adaptação dinâmica.

Nesse sentido, com o objetivo de contribuir com uma proposta de arquitetura inovadora, o *framework* Cosmos define uma nova abordagem para o processo de desenvolvimento de sistemas multimídia distribuídos, onde são considerados os requisitos apresentados na Seção 2.3.

A discussão sobre o primeiro requisito enumerado, conectividade em plataformas distribuídas heterogêneas, está presente nos principais sistemas de *middleware* atuais, uma vez que a maioria deles considera o modelo de referência RM-ODP. As recomendações do modelo RM-ODP foram observadas em todas as etapas do processo de desenvolvimento da proposta do Cosmos, sobretudo, no modelo de interconexão de componentes, que trata interconexões envolvendo componentes locais e remotos, e que podem potencialmente usar diferentes tecnologias de forma transparente.

O presente capítulo inicia introduzindo o modelo arquitetural do Cosmos e, em seqüência, apresenta uma visão funcional com as interfaces dos componentes da arquitetura e os relacionamentos entre os mesmos. O modelo arquitetural descreve a estrutura global do sistema envolvendo os principais componentes, suas responsabilidades e respectivos relacionamentos. Adicionalmente, a abordagem procura relacionar as interfaces com o suporte aos requisitos identificados na Seção 2.3. Uma visão do processo de integração destes elementos é apresentada no capítulo através da descrição

de alguns diagramas de seqüência UML. No capítulo são discutidas as arquiteturas definidas para gerenciamento de QoS e de interconexão de componentes. No Cosmos, questões relacionadas com os aspectos não-funcionais são tratadas no nível da configuração da aplicação, fora dos componentes que compõem a lógica da aplicação, como ocorre no *framework* EJB [112]. Para tratar estas questões, o Cosmos explora os conceitos de metadados e metacomponentes de forma semelhante a outros trabalhos como, por exemplo, o OpenORB [20] [130] [134].

4.1 Modelo Arquitetural

Como mencionado nos capítulos anteriores, o *framework* Cosmos foi proposto para dar suporte à configuração e gerenciamento de recursos e componentes de aplicações na camada de *middleware* de uma variedade de sistemas multimídia distribuídos. Para isso, Cosmos define um modelo de componentes abstrato (*CosmosComponent*) que incorpora as funcionalidades básicas dos elementos arquiteturais do sistema, podendo ser estendido, instanciado e especializado para contextos particulares.

4.1.1 Principais Elementos

Como o desenvolvimento baseado em componentes ainda é uma tecnologia nova para a engenharia de *software*, é comum encontrar várias abordagens conceituais para definições de componentes, *frameworks*, modelo arquitetural e modelo funcional. Assim, para evitar ambigüidade na apresentação do modelo, este trabalho decidiu utilizar os conceitos definidos por Szypersky [27] [28]. Desta forma, componentes no Cosmos são unidades de composição com interfaces especificadas contratualmente num contexto de dependências explícito. Assim, um componente de *software* pode ser implantado independentemente das partes que o compõem. O *framework* define um conjunto de interfaces e regras para gerenciamento e interação entre os diversos componentes do sistema.

Como base para a abordagem arquitetural, o Cosmos define um componente abstrato, denominado *CosmosComponent*, com as interfaces básicas que devem ser providas obrigatoriamente por qualquer componente concreto do *framework*. A Fig. 4.1 apresenta um modelo simplificado que descreve os principais elementos arquiteturais do Cosmos e seus relacionamentos.

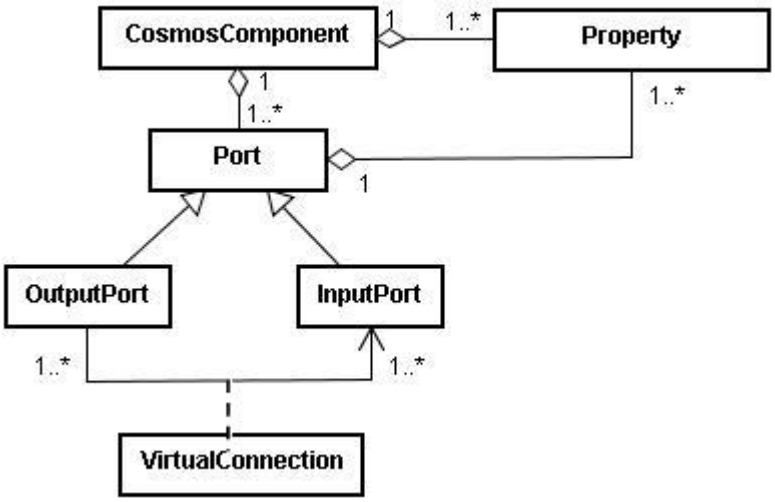


Fig. 4.1 Principais elementos arquiteturais do Cosmos

O conceito de recurso lógico [7] [8] [148] está presente no Cosmos com a denominação de *VirtualResource*. *VirtualResources* são componentes concretos, definidos no Cosmos para representar recursos da plataforma (*hardware* ou *software*), estando normalmente associados a processamento de fluxos de mídia. *VirtualResources* são portanto componentes específicos da plataforma; no entanto, deve-se observar que eles devem implementar todas as interfaces definidas para o componente abstrato *CosmosComponent*. Além destas interfaces, eles podem definir outras, de modo a especializá-los de acordo com as funcionalidades específicas que cada recurso representa. Por exemplo, um *VirtualResource* pode definir novas interfaces para controlar a sincronização, levando em consideração o comportamento do tipo do recurso.

O Cosmos explora o conceito de portas para tratamento de fluxos multimídia. Portas são pontos por onde elementos de um fluxo podem entrar ou sair de um componente. Cada porta estabelece um ponto de conexão que pode ser usado para conectar um componente associado com o seu ambiente de execução (um ou mais componentes).

Um componente concreto (*VirtualResource*) pode ter várias portas associadas, onde cada porta é caracterizada por um tipo que denota as características dos dados que por ela fluem. De forma a suportar o tratamento de vários estilos de interconexão (topologias), Cosmos define diferentes tipos de interfaces providas pelas portas, cada uma suportando operações com semânticas distintas, como por exemplo, interface com operações para portas do tipo *OutputPort* (porta de saída) e interface com operações para portas do tipo *InputPort* (porta de entrada). Desta forma, uma porta *outputPort* deve

implementar uma interface que provê suporte para comunicações que fluem do componente em direção ao ambiente (com comportamento *unicast* ou *multicast*), enquanto portas *inputPort* devem receber fluxos do ambiente, em direção ao componente. No modelo simplificado da Fig. 4.1 foi introduzido um componente *Port* para tratar os conceitos e comportamentos comuns a portas. No Cosmos, portas também são componentes, ou seja, portas possuem uma interface de configuração que pode ser usada para definir, monitorar e/ou alterar dinamicamente propriedades associadas às mesmas (como, por exemplo, os parâmetros de QoS). Os tipos concretos de componentes *OutputPort* e *InputPort* definem respectivamente comportamentos para portas de saída e de entrada.

Na Fig. 4.1 foram exemplificados os papéis de portas de entrada e saída sem considerar aspectos de topologias de conexão. No modelo definido pelo Cosmos, componentes locais ou remotos podem ser interconectados através de ligações entre portas através de um componente denominado conexão virtual (*VirtualConnection*). Um componente *VirtualConnection* abstrai o conceito de fluxo de dados multimídia entre portas, provendo uma interface bem definida para configuração e gerenciamento das respectivas ligações. Na verdade, a transferência dos dados do fluxo não é realizada por componentes *VirtualConnections*; cabe à conexão virtual apenas criar e configurar os recursos e componentes do *middleware* que efetivamente realizam a comunicação adequada. Conexões devem envolver pares de portas de entrada e saída e, conforme está descrito na Fig. 4.1, o modelo suporta conexões do tipo várias portas de saída para várias portas de entrada. Os aspectos relacionados com o modelo para gerenciamento de conexões são tratados na Seção 4.2.4.

Considerando que reusabilidade é um princípio fundamental, a construção de componentes Cosmos (aplicações, ou mesmo recursos virtuais comuns) permite o uso de técnicas de composição, ou seja, o uso de uma coleção de outros recursos virtuais pré-fabricados, ou mesmo de aplicações previamente desenvolvidas. Para prover condições que propiciem o incremento no nível de reusabilidade, deve-se preservar a manutenção das funcionalidades definidas nas interfaces do componente abstrato *CosmosComponent.*

Outro aspecto importante do *framework* é o suporte ao gerenciamento de QoS. Neste contexto, o *framework* define um modelo de QoS que permite realizar o monitoramento e eventual ajuste de comportamento dos componentes, bem como de conexões entre componentes.

No decorrer do capítulo, os elementos que compõem os vários componentes e modelos do Cosmos são descritos.

4.1.2 Modelo de Componentes

A tecnologia atual tem oferecido várias soluções de suporte ao uso de componentes de *software*. Para prover flexibilidade aos desenvolvedores, de forma que eles possam desenvolver suas aplicações usando a integração de soluções oriundas de diferentes tecnologias, é necessário definir um modelo de componentes neutro que trate esta diversidade de forma independente. Com este objetivo, foi definido um modelo de componentes próprio no Cosmos. O uso deste modelo permite desenvolver componentes de uso geral, provendo assim níveis elevados de reusabilidade. Com esta finalidade, foram definidas algumas interfaces, chamadas no Cosmos de interfaces básicas, que devem ser providas, em caráter obrigatório, por qualquer componente do *framework*. Estas interfaces foram definidas no escopo do componente abstrato *CosmosComponent*. Dentre as interfaces básicas definidas, dá-se especial destaque para as interfaces de propriedades (*properties*). Vale a pena lembrar que o conceito de propriedades está presente em outras abordagens, como por exemplo, no PREMO [7], *A/V Streams* [8] e no CM-tel [19].

Uma propriedade pode ser definida dinamicamente como um par de elementos (chave, valor), onde a chave corresponde a uma cadeia de caracteres (*string*), usada por exemplo, para identificar uma característica do componente, e valor pode ser uma instância de qualquer tipo de dados, utilizada para descrever a característica. Uma simplificação adotada nas interfaces definidas para propriedades no Cosmos, foi a de tratar valores do tipo *string* e listas de *strings*. Usando o conceito de propriedades, dispositivos heterogêneos podem ser configurados e especializados de forma uniforme através da definição de valores específicos de propriedades.

A abordagem de prover interfaces básicas para gerenciamento de recursos também está presente em outros trabalhos recentes [20] [100]; no entanto, no modelo de componentes de Cosmos, não existe a idéia de herança explorada nestes trabalhos.

Como mencionado, o componente abstrato *CosmosComponent* trata as interfaces e conceitos básicos que devem ser providos por quaisquer componentes do *framework*. Esta abordagem de organizar e separar os elementos do modelo em diferentes estruturas (elementos básicos e elementos específicos), além de facilitar a apresentação e a compreensão do *framework*, ajuda na elaboração de projetos e implementações de sistemas de *middleware* baseados no Cosmos. As interfaces básicas do *framework* correspondem às operações para manipulação de propriedades e gerenciamento de recursos e ciclo de vida. As idéias usadas nas definições de operações de gerenciamento de recursos foram inspiradas nas propostas PREMO [7] e OpenORB [100] [130].

A capacidade de extensão do Cosmos está associada principalmente com a facilidade de definição de recursos virtuais com APIs especializadas. Esta facilidade dá ao Cosmos a caracterização de ser um *framework* genérico, podendo assim ser instanciado para diferentes tipos de aplicações multimídia. Estas APIs podem ser usadas por desenvolvedores ou por ambientes de suporte em uma variedade de dispositivos para controle, gerenciamento, distribuição e apresentação de uma diversidade de artefatos multimídia.

O conceito de propriedades provê também a base para configuração e negociação de QoS, constituindo-se num mecanismo que permite o *middleware*, por exemplo, guardar os valores dos parâmetros de configuração obtidos durante a fase de processamento da especificação da aplicação, bem como controlar a QoS na fase de execução.

As interfaces de propriedades provêem também suporte para realização de consultas e definição de características de componentes, constituindo-se em um mecanismo de suporte ao conceito de reflexividade. Assim, os componentes da aplicação podem eventualmente consultar e definir mudanças no comportamento do componente, através de operações das interfaces de propriedades.

Por exemplo, o configurador, ou a própria aplicação podem usar operações de consulta para obter um parâmetro com a indicação da QoS atualizada correspondente à operação do sistema (propriedade), para fins de negociação, ou para obter valores comuns de operação entre diferentes dispositivos.

As operações básicas definidas para manipulação de propriedades foram agrupadas em diferentes interfaces do componente abstrato *CosmosComponent*, conforme ilustrado na Fig. 4.2. Na seqüência, o texto faz algumas discussões acerca destas operações.

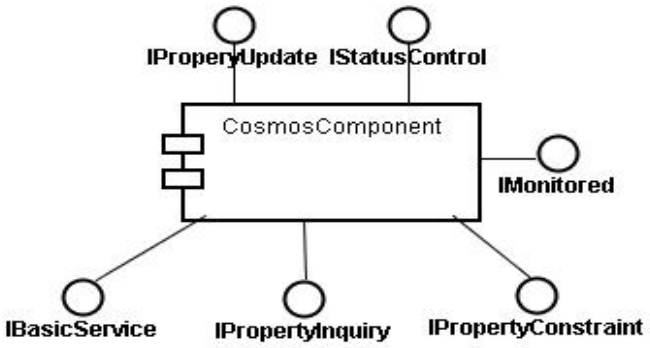


Fig. 4.2: Interfaces do Componente *CosmosComponent*

A interface *IBasicService* fornece operações para obtenção de informações básicas do componente, como por exemplo, as interfaces por ele providas. A Fig. 4.3 descreve as operações que foram definidas para a interface *IBasicService*.

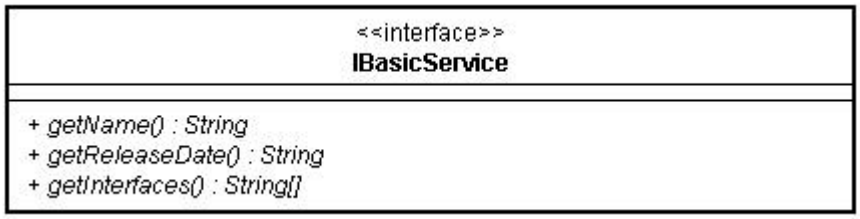


Fig. 4.3: Interface *IBasicService*

A operação *getName* fornece a identificação do componente e *getReleaseDate* a data de distribuição do mesmo; já a operação *getInterfaces* fornece como resultado a lista com a identificação de todas as interfaces do componente.

A interface *IPropertyInquiry* é definida para consultas de valores de propriedades. Uma descrição desta interface envolvendo suas operações é apresentada na Fig. 4.4.

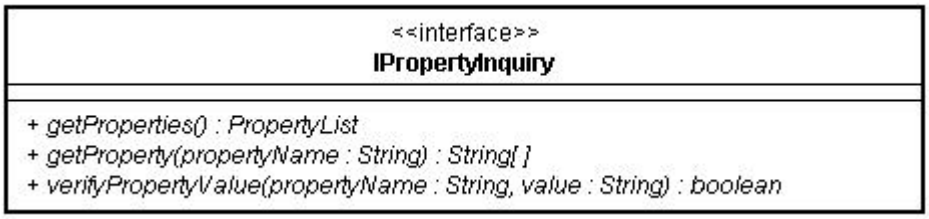


Fig. 4.4: Interface *IPropertyInquiry*

Uma propriedade pode ter um conjunto de valores possíveis. A operação *getProperties* retorna uma lista contendo todas as propriedades do componente, enquanto a operação *getProperty* retorna a lista de valores correspondente à propriedade indicada no parâmetro *propertyName*. A operação *verifyPropertyValue* é usada para verificar se a propriedade indicada suporta o valor passado como argumento.

A interface *IPropertyConstraint*, apresentada na Fig. 4.5, fornece a operação *setConstraint* que indica uma restrição de valor para efeito da configuração de uma determinada propriedade em uma instância do componente. Por exemplo, uma instância do componente pode restringir determinados valores de propriedades, indicando desta forma que estes valores não sejam considerados na operação do componente.

Fig. 4.5: Interface *IPropertyConstraint*

O projetista de um componente, que é o elemento responsável por descrever as características do seu comportamento, pode definir uma operação na interface funcional específica do mesmo que permita a manipulação simplificada de possíveis valores de propriedades; as interfaces funcionais específicas de um componente são as que apresentam maior interesse para a aplicação, pois elas determinam o comportamento "visível" da operação do componente.

As interfaces de propriedades que foram introduzidas no componente básico *CosmosComponent* são caracterizadas por definirem operações com semântica simplificada. Para permitir mudanças consistentes de configuração, o *middleware* deve supervisionar as mudanças verificando se a mudança é coerente com estado atual da plataforma. Adiante, nas Seções 4.2 e 4.3 são apresentados os modelos funcionais e de gerenciamento de recursos que dão suporte à realização de adaptação e reconfiguração dinâmicas com tratamento de consistência.

A interface *IStatusControl*, descrita na Fig. 4.6, oferece uma operação para gerenciamento e controle de mudança de estado do componente. Deve-se destacar que cada componente tem uma máquina de estados associada que determina os possíveis comportamentos e transições de estado válidas.

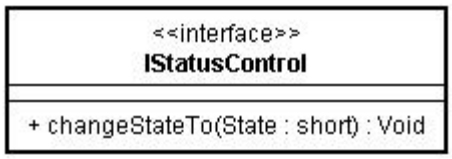


Fig. 4.6: Interface *IStatusControl*

A interface *IPropertyUpdate*, apresentada na Fig. 4.7, é utilizada para a atualização de valores selecionados de propriedades dos componentes quando houver eventuais alterações em decorrência, por exemplo, de negociações dinâmicas. Para isto, ela define a operação *changePropertyValue* onde são passados como parâmetros, o nome da propriedade e o novo valor. Também foram definidas nesta interface as operações *activate* e *deactivate*, que são utilizadas para ativação e desativação do recurso,

respectivamente. A operação *activate* é usada para informar que os recursos necessários à operação do componente foram obtidos e a operação *deactivate* informa que os mesmos devem ser liberados.

Fig. 4.7: Interface *IpropertyUpdate*

A interface *IMonitored* é utilizada para fins de gerenciamento de QoS. Neste sentido, ela é discutida adiante, na Seção 4.2.3, no contexto da arquitetura de gerenciamento de QoS.

4.1.3 *Framework* Arquitetural

Para dar suporte à configuração e gerenciamento de componentes, Cosmos define o *framework* arquitetural introduzido na Fig. 4.8, onde são destacados os seguintes elementos:

- ***ApplicationSpecification*:** descrição da aplicação envolvendo os componentes e respectivas conexões;
- ***Configurator*:** componente responsável por iniciar, configurar e coordenar o processo de gerenciamento dos componentes do *middleware* e das aplicações;
- ***ApplicationProxy*:** mantém na camada de *middleware* uma descrição dos componentes básicos de uma aplicação em execução, sendo responsável pelas operações de manipulação de propriedades desses componentes. Cada operação do componente *ApplicationProxy* chamada é analisada com relação à coerência, e em seqüência, uma ação associada é acionada no correspondente componente básico da aplicação;
- ***Factories*:** responsáveis pela criação de componentes (portas, serviços, recursos e demais componentes do *framework*);
- ***Resources*:** representação de recursos (*hardware* e *software*); e
- ***Services*:** permite incluir novas funcionalidades, como por exemplo, serviço de repositório para obtenção de fábricas e componentes.

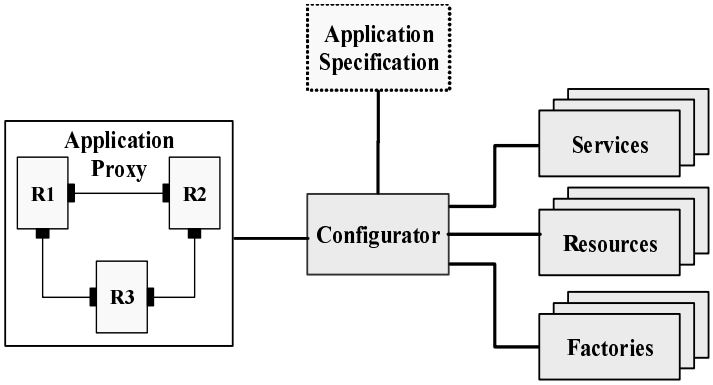


Fig. 4.8: Arquitetura do Cosmos

Como elemento central do *framework*, o *Configurator* é um componente crítico, responsável pela iniciação, configuração e gerenciamento dos recursos do sistema, dos componentes do *middleware* e da aplicação.

Cada aplicação deve ser definida através de uma especificação (*Application Specification*) – descrição envolvendo os recursos e os elementos da arquitetura -, bem como os vários requisitos de QoS necessários para a sua execução. Desta forma, uma linguagem de especificação deve prover as características de uma ADL (*Architecture Description Language*) [149]. Fazendo uma analogia com os modelos de servidores de componentes comerciais atuais, esta especificação corresponde aos descritores de implantação (*deployment descriptions*) [21].

Para prover suporte a aspectos de reusabilidade e interoperabilidade, configurações devem suportar a composição de aplicações envolvendo componentes com uma variabilidade de características, tecnologias, propriedades e dependências. A distribuição das aplicações e componentes envolve níveis de complexidade elevados com relação a implantação e configuração dos componentes em dispositivos heterogêneos da plataforma. Devido a isso, a implantação, configuração e ativação de uma aplicação é uma atividade sujeita a erros decorrentes de potenciais incompatibilidades de integração. Para tratar estas questões, muito esforço tem sido feito na busca de um modelo padronizado para empacotamento de componentes e descrição de parâmetros de configuração, implantação e instanciação dos mesmos. As soluções atualmente utilizadas para descrever estes elementos são baseadas em XML e empregam intensamente os conceitos de metadados para a descrição do processo. Como resultado de um esforço para resolver este problema, a OMG apresenta uma recomendação denominada RAS (*Reusable Asset Specification*) [150]. Esta recomendação define uma coleção de modelos e padrões para a descrição de diferentes tipos de componentes de *software* reusáveis. Assim,

plataformas com suporte a *frameworks* de componentes precisam fornecer ferramentas para interpretar estas especificações de forma a proceder a instanciação e a configuração da aplicação.

No Cosmos, o *Configurator* é responsável pelo gerenciamento do ciclo de vida dos diversos componentes. Ele realiza operações de configuração, inspeção, negociações e ajustes dinâmicos de propriedades associadas a recursos virtuais e fluxos multimídia envolvidos na especificação.

A descrição de cada aplicação, conforme o modelo definido no Cosmos, é representada no *middleware* por um componente denominado *ApplicationProxy*. O principal papel deste componente consiste em construir e manter atualizados os metacomponentes que representam os componentes do sistema e da aplicação. Metacomponentes descrevem as propriedades e os estados dos componentes associados durante o ciclo de vida da aplicação. As interfaces definidas no componente *ApplicationProxy* dão suporte ao conceito de reflexividade, conceito este tratado de forma semelhante à abordagem utilizada em [100].

A denominação *ApplicationProxy* é um neologismo criado no Cosmos para designar um componente de *software* que não constitui-se diretamente na aplicação, mas que facilita o suporte à consistência em processos de adaptação envolvendo sistemas com níveis diversificados de requisitos, plataformas, tecnologias e propriedades. Cada operação associada a uma mudança dinâmica de propriedades de um componente precisa ser encaminhada ao componente *ApplicationProxy*. Este componente verifica a coerência da operação, atualiza a descrição da arquitetura e repassa a alteração (mudança de valores de propriedades) para os respectivos componentes afetados. Uma solução que também usa a denominação *proxy* no contexto de um *middleware* adaptativo foi descrita por Moreira [151] [152]. No contexto da tecnologia de componentes, a denominação *Proxy* é usada por Völter [21] para denotar referências a componentes virtuais. Na abordagem de Völter, clientes fazem chamadas a operações de componentes virtuais e o suporte provido pelo *contêiner* se encarrega de resolver a ativação do componente físico associado e repassar para ele a chamada da operação.

O *framework* Cosmos representa os recursos requeridos da aplicação no componente *ApplicationProxy* usando um grafo com os metacomponentes que descrevem os recursos. No Cosmos, metacomponentes são basicamente repositórios de metadados (propriedades) associados aos recursos, dando suporte para identificação, localização e endereçamento dos correspondentes componentes e respectivas interfaces, entre outros mecanismos disponibilizados. Metadados são usados principalmente para fins de configuração e adaptação. Os conceitos de metacomponente e metadado têm sido explorados em vários trabalhos relacionados à adaptação dinâmica [89] [100]. Porém, grande parte

destes trabalhos define interfaces de baixo nível, como por exemplo, a especificação de operações para manipulação de escalonamento e gerenciamento de *threads* [15].

De acordo com o modelo arquitetural do Cosmos, aplicações e componentes são descritos usando uma linguagem de especificação, conforme mencionado. O objetivo desta linguagem consiste em gerar metacomponentes que descrevem a configuração dos componentes da arquitetura. Para isto, são usadas informações definidas estaticamente, obtidas da própria especificação, ou informações definidas dinamicamente, obtidas através de consultas à API de reflexividade, ou de processos de interação com a própria a aplicação. A especificação da sintaxe para esta linguagem, bem como a sua implementação é responsabilidade do *middleware*. Poderá ocorrer inclusive que um *middleware* defina mais de uma sintaxe. Entre outros elementos, a especificação deve indicar os componentes envolvidos na aplicação, as dependências para efeito de composição, as propriedades dos componentes e das portas, as conexões entre componentes e os parâmetros de QoS associados aos componentes e às conexões.

Para a construção de uma aplicação, a especificação indica os componentes, suas dependências e as propriedades providas e requeridas dos mesmos. Usando estas informações, o *middleware* localiza os componentes em serviços de diretórios de modo a poder instanciá-los. Os elementos do *middleware* e da aplicação podem obter as interfaces disponibilizadas por eles, realizando chamadas à operação *getInterfaces* da interface *IBasicService*, conseguindo assim as referências para as interfaces por ele providas.

4.1.4 Modelo para Descrição de Aplicações

Conforme mencionado, em plataformas Cosmos o *Configurator* é o componente responsável por interpretar especificações, ou seja, por obter os dados (metadados) que descrevem as aplicações. Estes metadados devem ser mantidos durante o ciclo de vida das aplicações para serem usados dinamicamente em eventuais reconfigurações. Neste contexto, levando em consideração algumas experiências desenvolvidas anteriormente [31] [77] [154], foi definido um modelo para descrição dos conceitos e elementos do *framework* arquitetural apresentados na Seção 4.1.3. Um metamodelo em UML que descreve os elementos deste modelo é apresentado na Fig. 4.9. Considerando este modelo, uma linguagem de especificação de aplicações e componentes Cosmos é apresentada no Apêndice A.1 usando uma notação baseada em EBNF [155].

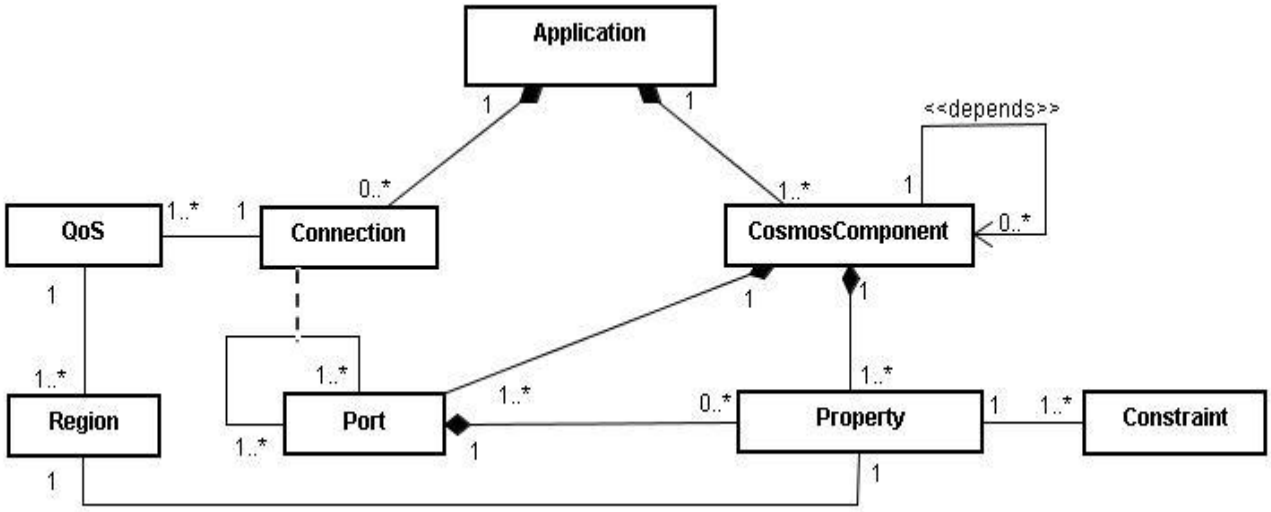


Fig. 4.9: Modelo para definição dos metacomponentes de uma especificação

Como pode ser observado, os elementos descritos na Fig. 4.9 contemplam os componentes do modelo arquitetural introduzido na Seção 4.1.3. Adicionalmente, a figura apresenta novos elementos arquiteturais para tratar requisitos relacionados a gerenciamento de QoS. O objetivo de uma especificação (descrição de uma aplicação) consiste em definir uma hierarquia de metadados a ser reconhecida e processada pelo *Configurator*. O princípio estabelecido no Cosmos é que, para cada componente arquitetural do *framework* definido, deve existir um metacomponente que descreve as características do respectivo componente.

Para instanciar uma aplicação, o *parser* da linguagem de especificação obtém os metadados da especificação de acordo com o esquema de processamento descrito na Fig. 4.10. Neste esquema, o papel do *parser* consiste em mapear a descrição (metaprograma) em propriedades de metacomponentes que descrevem as características dos componentes da aplicação e da própria plataforma. A definição de propriedades no contexto da especificação está associada aos conceitos explorados nas definições das interfaces de propriedades do Cosmos apresentadas na Seção 4.1.2. O conceito de propriedades freqüentemente é usado na literatura para tratar aspectos não-funcionais do sistema, diferentemente do conceito tradicional de atributos, que normalmente está associado a definições de parâmetros funcionais, tipicamente utilizados na instalação e instanciação de componentes. O uso de propriedades está associado a intormações para configuração e ajuste dinâmicos de componentes e de instâncias de componentes.

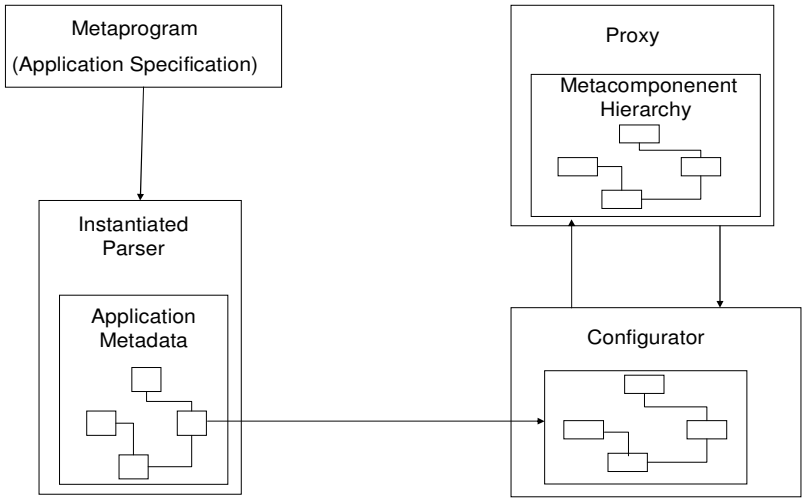


Fig. 4.10**:** Esquema de processamento de uma especificação

Como pode ser observado na Fig. 4.10, o *parser* instanciado está separado do *Configurator*. Isto permite incorporar diferentes linguagens e ferramentas de apoio à especificação de aplicações. O elemento que dá a base para esta diversidade é o modelo de metacomponentes (Fig. 4.9), que funciona como uma linguagem intermediária padrão para os *parsers* se comunicarem com o *Configurator*.

Os metadados de uma aplicação, de acordo com a Fig. 4.9, contêm listas de componentes e conexões, sendo compostos por um ou mais *CosmosComponentMetadata* - elemento que descreve um componente da aplicação -, e por nenhum ou muitos *ConnectionMetadata*, elemento que descreve uma conexão.

A descrição de um componente Cosmos (*CosmosComponentMetadata*) pode definir uma ou mais propriedades (*PropertyMetadata*) e uma ou várias portas (*PortMetadata*). A descrição da aplicação pode também conter restrições (*ConstraintMetadata*) associadas a propriedades dos respectivos componentes. No diagrama da Fig. 4.9 mostra-se também que um Componente Cosmos pode estar relacionado com outros, indicando por exemplo relações de dependência.

Restrições são representadas e manipuladas de forma semelhante a propriedades. Com o objetivo de não sobrecarregar a Fig. 4.9, foram omitidas algumas relações envolvendo os componentes arquiteturais com *Property* e *Constraint*. Deve-se lembrar que o relacionamento explicitado entre estes componentes e *CosmosComponent* também se repete com os demais metacomponentes.

Um componente *Port* contém as propriedades que descrevem a respectiva porta Na especificação de uma aplicação, a descrição de uma porta (*PortMetadata*) também pode ter restrições associadas às suas propriedades. Uma abordagem detalhada sobre o modelo de portas é apresentada adiante, no modelo de interconexão descrito na Seção 4.2.4.

Conexões (*Connection*) são descritas em uma especificação por componentes *ConnectionMetadata*; uma conexão envolve relacionamentos entre portas, devendo existir pelo menos uma porta de origem e uma de destino, conforme o modelo introduzido na Fig. 4.1.

Uma conexão também pode conter componentes que descrevem requisitos de QoS – representados na Fig. 4.9 por componentes *QoS*. Estes componentes poderão ser usados para gerenciamento de parâmetros de QoS associados à conexão [83] [84]. Para o modelo de QoS descrito adiante, elementos *QoSMetadata* contêm as regiões de adaptação (faixas de valores de propriedades), consideradas em função dos valores de propriedades observados dinamicamente. Uma abordagem sobre o modelo de QoS definido para o Cosmos é apresentada e detalhada na Seção 4.2.3. Problemas relacionados com a QoS estão freqüentemente presentes na realização de comunicação distribuída. Em função disso, a arquitetura atual do Cosmos aborda esta questão no contexto de conexões entre componentes. Embora Cosmos não tenha realizado experiências usando o modelo de QoS em outros componentes do *framework*, a abordagem é aplicável em adaptações envolvendo outros aspectos, como por exemplo, a carga de CPU.

A abordagem definida na Fig. 4.10 provê mais flexibilidade em relação ao modelo baseado em XML proposto anteriormente [76] [77]; esta flexibilidade decorre da abstração da escolha de uma linguagem de descrição específica, permitindo introduzir novas linguagens além de XML, sem que isto venha a trazer impactos na implementação do componente *Configurator*. Assim, o desenvolvedor tem a possibilidade de utilizar a linguagem de descrição que ele escolher ou projetar, devendo para isso definir os seus respectivos *parsers*, os quais deverão mapear as estruturas da linguagem para o modelo de metacomponentes do Cosmos. Metadados são obtidos pelo *Configurator* e introduzidos nos correspondentes metacomponentes.

4.1.5 Modelo Básico para Criação de Componentes

O Cosmos estabelece que a tarefa de criação de componentes deve ser realizada por componentes denominados *Factories*. Conforme a descrição da Fig. 4.11, um componente *Factory*

tem, além das interfaces definidas no componente *CosmosComponent*, a interface *IFactory* com uma operação para criar componentes.

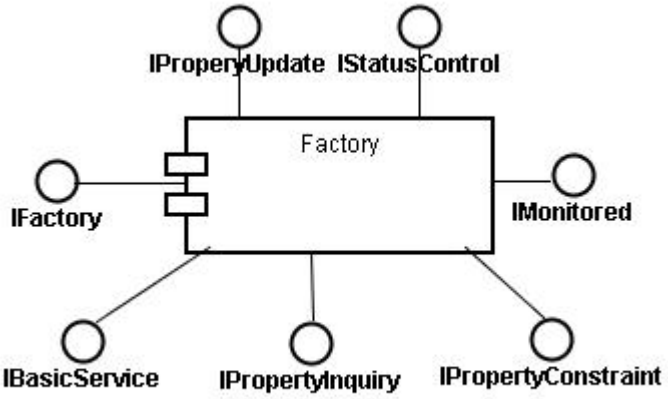


Fig. 4.11: Interfaces do componente *Factory*

A Fig. 4.12 apresenta a interface *IFactory* com a descrição da operação *create*; esta operação é responsável por criar componentes de acordo com os metadados do componente passados como parâmetro.

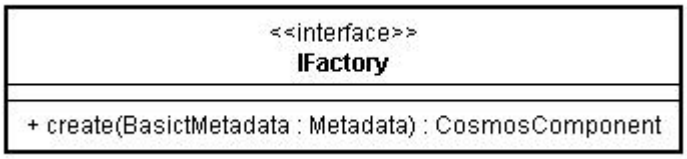


Fig. 4.12: Interface *IFactory*

No exemplo da Fig. 4.13, apresenta-se um modelo básico de um processo para criação de um componente. Neste processo, o *Configurator* obtém os metadados que descrevem o respectivo componente no *ApplicationProxy* (operação *getMetadata*) e identifica a fábrica adequada de acordo com estes metadados. Para dar características de generalidade à fábrica, o parâmetro passado é de um tipo básico. A fábrica indicada recebe então a solicitação para criação do componente (mensagem *create*) e os respectivos metadados passados no parâmetro. Após a instanciação do componente, o mesmo é registrado no *ApplicationProxy*, de forma que o *Configurator* tenha acesso às informações do componente. As operações *getMetadata* e *registerComponent* são denominações que têm como objetivo simplificar a descrição que envolveria várias chamadas associadas ao processo. Este diagrama apresenta, de maneira simplificada, um processo genérico de criação de um componente no *framework*

Cosmos; no processo não são considerados, por exemplo, aspectos como localização do componente. O foco da Fig. 4.13 está concentrado na descrição das ações realizadas pelo *Configurator* no instante da criação de um componente.

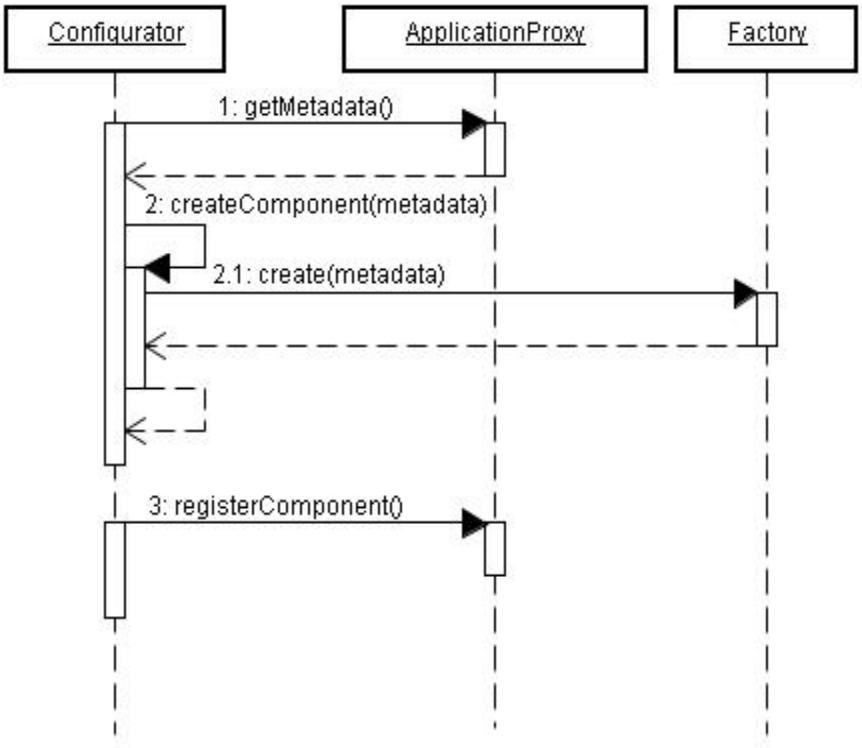


Fig. 4.13**:** Processo básico para criação de um componente no Cosmos

4.2 Modelo Funcional

Os modelos arquitetural e de metacomponentes definidos nas seções anteriores constituem-se em elementos importantes para o suporte aos requisitos de sistemas multimídia distribuídos abertos. A seguir, a Fig. 4.14 apresenta uma visão funcional integrada dos elementos destes modelos. Na seqüência, o texto descreve as principais interfaces do modelo funcional estabelecendo uma associação entre elas e os requisitos relacionados no Capítulo 2:

- Suporte à configuração e reconfiguração;
- Suporte à adaptação;
- Suporte à QoS;
- Suporte à conectividade em plataformas distribuídas heterogêneas.

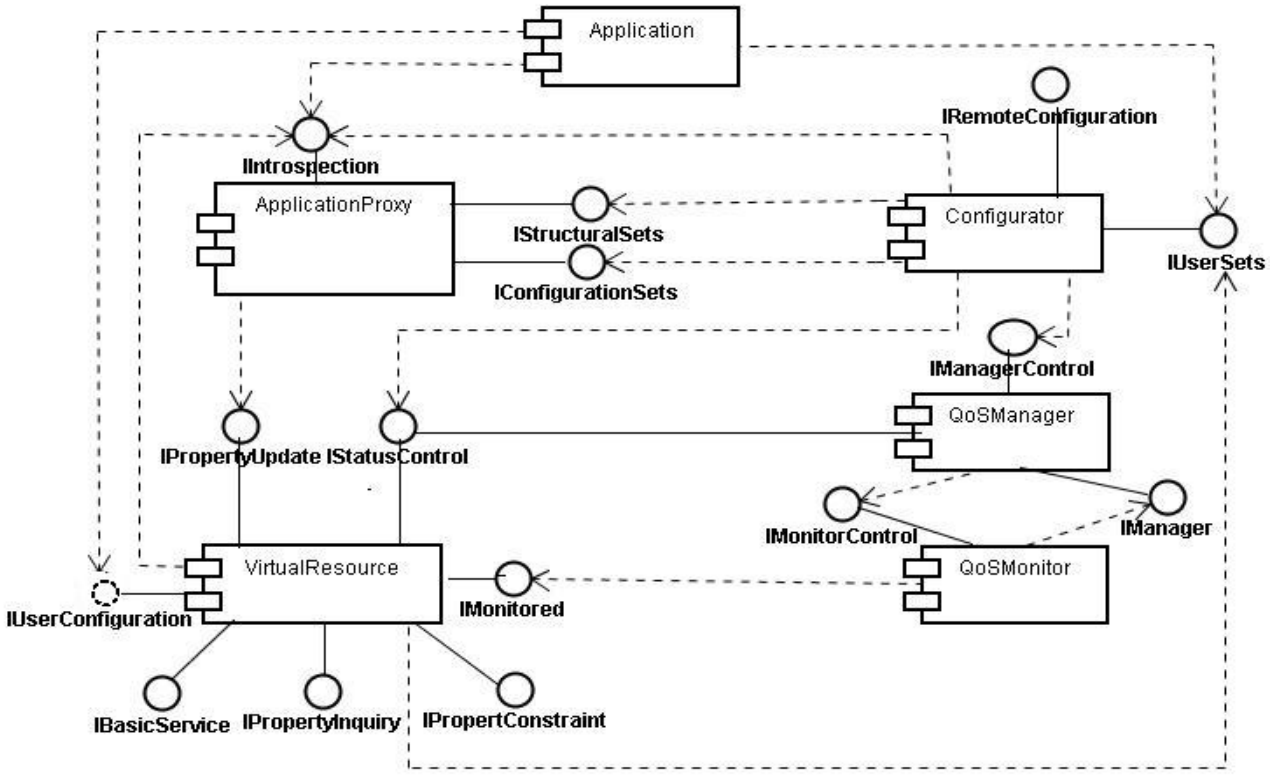


Fig. 4.14**:** Modelo com a Arquitetura Funcional do Cosmos

4.2.1 Suporte para Configuração e Reconfiguração

Como descrito na Seção 4.1.4, o *parser* deve gerar os metacomponentes que descrevem os componentes da aplicação a partir da especificação fornecida. Navegando na estrutura destes metacomponentes, o *Configurator* pode executar as operações correspondentes à configuração do *middleware* e da aplicação.

O processo de configuração consiste em consultar e processar os metadados inseridos nos metacomponentes, de modo a saber quais recursos virtuais devem ser instanciados e suas respectivas propriedades. Metacomponentes no Cosmos são manipulados por operações definidas no componente *ApplicationProxy*.

Como o Cosmos dá suporte ao desenvolvimento e execução de aplicações distribuídas, componentes de uma aplicação poderão ser instanciados em diferentes domínios (espaços de endereçamento). Assim, considerando que uma determinada fábrica de recursos virtuais com capacidade para instanciar um recurso virtual requerido na especificação tenha sido selecionada e

localizada, cabe ao *Configurator* acionar esta fábrica para instanciar o referido recurso virtual, localmente ou remotamente, de forma que posteriormente possa realizar operações de configuração e eventuais reconfigurações.

Caso o recurso em questão precise ser instanciado em outro domínio, o *Configurator* se reportará ao *Configurator* remoto responsável por gerenciar os recursos daquele domínio através da interface *IRemoteConfiguration* (Fig. 4.14), que comandará o processo de instanciação do referido recurso virtual a partir dos metadados definidos. Deve-se ressaltar, no entanto, que os metadados do recurso virtual a ser instanciado remotamente residirão no componente *ApplicationProxy* da aplicação; o componente *ApplicationProxy* reside no mesmo domínio em que se encontra o *Configurator* responsável por processar a especificação. O *Configurator* que tem esta responsabilidade é denominado no Cosmos como *Configurator* mestre da correspondente aplicação. Os demais *Configurators* envolvidos com o gerenciamento de recursos virtuais remotos de uma aplicação são denominados *Configurators* escravos.

Configurators são portanto elementos que gerenciam em cada domínio os componentes das várias aplicações ali instanciados, existindo um *Configurator* por domínio. Um *Configurator* pode gerenciar várias aplicações, porém cada aplicação possui apenas um *Configurator* mestre que é responsável por comandar o gerenciamento desta aplicação. Cada aplicação tem um grupo de metacomponentes representado e manipulado por um componente *ApplicationProxy*, sob o controle e supervisão do respectivo *Configurator* mestre. Cabe ao *Configurator* mestre de uma aplicação manter a consistência dos recursos do sistema alocados à aplicação.

No componente *ApplicationProxy* da aplicação são mantidas referências simbólicas para os recursos virtuais remotos, de forma a prover suporte para realização de *binds* que permitam o *Configurator* mestre acessar os correspondentes recursos virtuais para atualizar, por exemplo, valores de propriedades decorrentes de uma adaptação ocorrida durante a execução da aplicação.

Uma vez que o *Configurator* é o elemento responsável pela negociação e instanciação de componentes, deve existir em um sistema distribuído uma interação entre os *Configurators* envolvidos. Esta interação envolve questões sobre a alocação de recursos, negociação de propriedades e instanciação dos componentes remotos. Uma vez que o *Configurator* é quem controla seu espaço de endereçamento, somente ele pode validar uma determinada configuração. A validação neste contexto, refere-se a análise da viabilidade da instanciação.

O nível de interação entre *Configurators* remotos se limita a pedidos para criação e remoção de componentes, bem como para validação de requisição de recursos (verificação se o pedido é coerente com a disponibilidade de recursos da plataforma). Neste caso, o *Configurator* "escravo" gerencia os componentes locais de uma aplicação em que o *Configurator* mestre estiver remoto, acessando remotamente o *ApplicationProxy* (através de um *bind* para a interface *IIntrospection*).

A seguir, a Fig. 4.15 descreve as operações da interface *IRemoteConfiguration*. A operação *createRemoteComponent* é usada para solicitar a criação de um componente remoto; para isto, devem ser passados para o *Configurator* escravo os metadados do componente a ser criado e uma referência simbólica do *ApplicationProxy* que mantém estes metadados. De forma complementar, a operação *destroyComponent* é usada para solicitar a remoção de um componente remoto, sendo portanto necessário passar uma referência simbólica do referido componente.

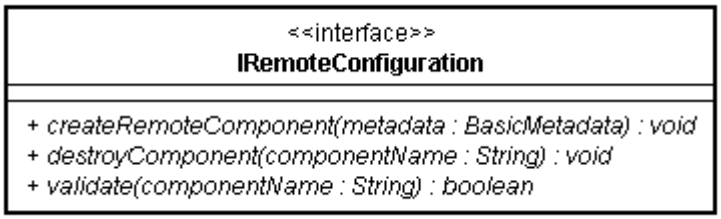


Fig. 4.15: Interface *IRemoteConfiguration*

A configuração de propriedades, mesmo em componentes remotos, é tarefa do *Configurator* mestre. No entanto, apesar do *Configurator* mestre ter a responsabilidade de gerenciar todos os recursos da aplicação, ele não pode, a priori, assegurar os recursos de um componente remoto, cujo gerenciamento é de responsabilidade do correspondente *Configurator* remoto. Assim, o *Configurator* mestre precisa se reportar a este *Configurator* remoto através da operação *validate* da interface *IRemoteConfiguration*, de modo que ele possa verificar se a configuração indicada (correspondente aos metadados configurados no *ApplicationProxy*) é coerente com a disponibilidade de recursos da plataforma local. Como mencionado, a validação é limitada a análise de viabilidade da configuração. Caso a configuração seja considerada viável, o *Configurator* escravo já deixa o recurso alocado. Caso contrário, o *Configurator* mestre muda a configuração da aplicação e tenta validar novamente, assim continuando até concretizar o processo ou esgotar as possíveis alternativas de configuração.

A decisão de usar a operação de validação apenas em requisições remotas, deve-se ao fato que localmente a definição de um valor de propriedade significa que o *Configurator* assegura o recurso relacionado com propriedade que ele mesmo negociou, ao passo que para um componente remoto, uma

atribuição de valor de propriedade no *ApplicationProxy* significa apenas uma proposição a ser validada pelo *Configurator* remoto. Para dar início a uma aplicação, é então necessário que todos os componentes remotos tenham suas indicações de recursos validadas.

Um *ApplicationProxy* também é um componente Cosmos, portanto deve também prover as interfaces básicas. Assim, ele também possui as operações *activate* e *deactivate* definidas na interface *IPropertyUpdate* (Fig. 4.7). O papel da operação *activate* no *ApplicationProxy* consiste em propagar, para todo o grupo de componentes da aplicação, a indicação de que os recursos foram negociados e estão passíveis de serem utilizados, colocando então em execução a aplicação. De forma complementar, a operação *deactivate* tem o papel de desativar os mesmos.

Como o modelo de componentes do Cosmos aceita a construção de componentes a partir de outros, o conceito de composição também é válido para *ApplicationProxies*; ou seja, pelo modelo do *framework* Cosmos, uma aplicação pode ser considerada como um componente reusável em outras aplicações. Para isto, deve-se disponibilizar serviços de persistência para os metacomponentes do *ApplicationProxy*.

Para consultar valores de propriedades de recursos virtuais para fins de alocação de recursos, um *Configurator* escravo usa as operações da interface *IIntrospection* do componente *ApplicationProxy*. *Configurators* escravos normalmente realizam estas consultas quando forem chamados para validar requisições de alocação de recursos. Conforme mencionado, se uma requisição não resultar em sucesso, cabe ao *Configurator* mestre mudar os valores de propriedades no correspondente *ApplicatiionProxy* e fazer uma nova chamada tentando realizar a alocação novamente, até conseguir a alocação ou esgotar as opções de configuração.

As interfaces que dão suporte à configuração (definição e alteração de valores de propriedades) são a *IStructuralSets* e a *IConfigurationSets*. Como a reconfiguração dinâmica está associada com ações de adaptação, as operações destas interfaces são descritas na próxima seção, de forma integrada com as interfaces de suporte à adaptação.

4.2.2 Suporte para Adaptação

A presente seção discute e apresenta as interfaces do Cosmos relacionadas com os requisitos de adaptação enumerados na Seção 2.3.3.

No modelo funcional, as características de reflexividade e adaptação são tratadas pelas interfaces *IStructuralSets, IConfigurationSets e IIntrospection.* Deve-se destacar que no modelo definido, as adaptações são realizadas de forma automática pelo *Configurator.* O modelo suporta a realização de adaptações originadas por necessidades de evolução do sistema, por eventos da aplicação, denominada no contexto da tese como adaptação reativa, ou por decisão interna do próprio *middleware* ao detectar, por exemplo, uma situação de sobrecarga temporária que venha a provocar uma eventual baixa de QoS. Este tipo de adaptação é denominado no texto como adaptação pró-ativa.

Para que a adaptação possa ser feita de forma consistente em ambos os casos, o *middleware* ou aplicação precisa realizar consultas sobre o estado atual do sistema. Para isso, utiliza-se a interface *IIntrospection*. Após a obtenção de dados atuais sobre o comportamento do sistema, o processo deve realizar operações de *matchings* para identificar valores comuns possíveis de propriedades. Por fim, a alteração de valores de propriedades - realizadas sob o controle do *Configurator* -, ocorre através das interfaces *IStructuralSets* e *IConfigurationSets*; a escolha de um valor adequado para alteração de uma propriedade deve ser realizada considerando valores coerentes entre os requisitos do sistema e o estado da plataforma, possivelmente identificados através da realização de operações de *matchings*.

Para facilitar a navegação da aplicação no grafo de metacomponentes definido no *ApplicationProxy* e manter o nível de consistência do sistema, cada componente *VirtualResource* pode prover interfaces específicas, como a interface *IUserConfiguration*. Esta interface está representada na Fig. 4.14 adotando um ícone tracejado como convenção para indicar que ela é específica. Portanto, não é preocupação do *framework* definir estas interfaces. Interfaces específicas devem definir operações de manipulação de propriedades limitadas, de forma que a aplicação só tenha acesso a operações de manipulação das propriedades que sejam relevantes ao seu interesse. Neste caso, as operações de consulta de propriedades originadas pela aplicação são repassadas para a interface *IIntrospection*, e as chamadas de operações para modificação de valores de propriedades, para a interface *IUserSets*. Assim, fica com o *Configurator* a responsabilidade de manter a consistência das modificações, bem como da sincronização da ativação da adaptação. Deve-se destacar que a aplicação pode acessar diretamente a interface *IIntrospection* de forma a obter uma visão global do estado do sistema. Esta permissão não oferece risco à manutenção da consistência, uma vez que a interface *IIntrospection* define apenas operações para consultas. De forma semelhante, a aplicação pode acessar diretamente a interface *IUserSets*.

A facilidade de estabelecer limites para a visibilidade do componente é uma tarefa do projetista do componente, pois presume-se que ele conhece o comportamento do mesmo e dos possíveis impactos decorrentes de mudanças de propriedades.

As operações definidas na interface *IStructuralSets*, descritas na Fig. 4.16, são usadas pelo *Configurator*, ou indiretamente, pela aplicação, para modificar a estrutura dos metacomponentes que representam a aplicação. As operações desta interface são as seguintes:

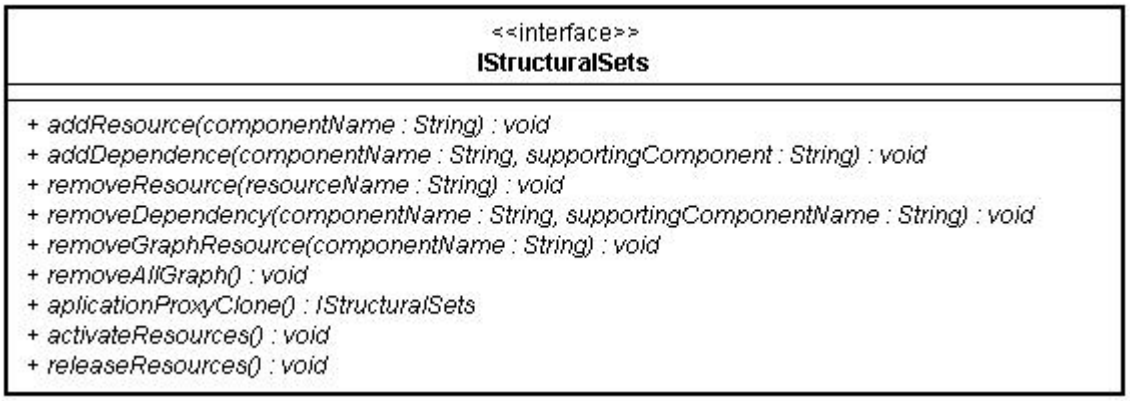


Fig. 4.16: Interface *IStructuralSets*

- *addResource*: adiciona ao *ApplicationProxy* o componente cuja identificação é passada no parâmetro.
- *addDependence:* usada para informar que o componente referente ao primeiro parâmetro depende do componente indicado no segundo.
- *removeResource*: remove o componente indicado no parâmetro, caso ele não dê suporte a outros componentes e não exista nenhuma dependência a ele associada. Para realizar remoções envolvendo grafos ou subgrafos de recursos, devem ser usadas as operações *removeGraphResource*, ou *removeAllGraphsResources*.
- *removeDependence:* usada para remover a dependência do componente referente ao primeiro parâmetro em relação ao indicado no segundo
- *applicationProxyClone*: cria um *ApplicationProxy* auxiliar; esta operação é ativada pelo *Configurator* na fase de negociações para preparar a configuração de uma adaptação, sem afetar a configuração corrente. Desta forma, mantém-se a aplicação em execução utilizando os metadados do *ApplicationProxy* original. Paralelamente, o *ApplicationProxy* recém-criado é

utilizado para definição de configurações temporárias, até que seja obtida uma configuração estável adequada à aplicação.

- *activateResources*: operação requisitada pelo *Configurator* com o objetivo de ativar os componentes da aplicação. O *ApplicationProxy*, que tem as referências de todos os componentes da aplicação, repassa para eles a solicitação, definindo uma ativação em bloco. Antes de executar a operação, o *ApplicationProxy* atualiza cada propriedade envolvida nos respectivos componentes da aplicação, através da interface *IActualizations*, de modo que eles possam operar corretamente. Este procedimento também deve ocorrer, durante a execução de uma reconfiguração, para então ele chamar a operação *changeStateTo* da interface *IStatusControl* (Fig. 4.6).
- *releaseResources*: operação requisitada pelo *Configurator* com o objetivo de desativar os componentes da aplicação, constituindo-se no procedimento inverso ao descrito na ativação.

A seguir, a Fig. 4.17 apresenta as operações da Interface *IConfigurationSets*, as quais dão suporte à realização de configuração e reconfiguração, consistindo nas operações de manipulação de valores de propriedades.

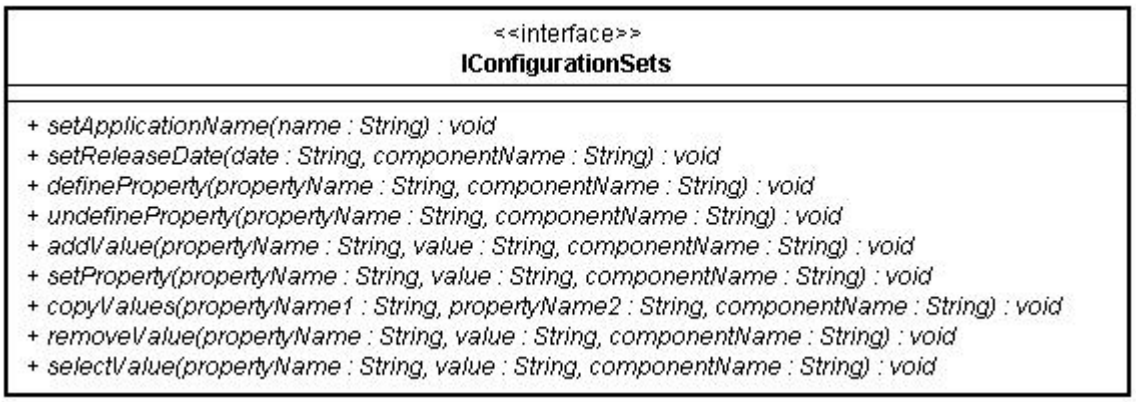


Fig. 4.17**:** Interface *IConfigurationSets*

A interface *IConfigurationSets* do componente *ApplicationProxy* contempla e dá suporte às operações para definição e alteração de propriedades. Uma breve discussão sobre as principais operações é apresentada nos parágrafos seguintes:

- A operação *setApplicationName* identifica o nome da aplicação e a operação *setReleaseDate* a data relativa à versão de distribuição do componente cujo nome é indicado.

- A operação *defineProperty* simplesmente identifica e cria uma propriedade, sem no entanto colocar valor. Isto tem como efeito apenas registrar que o componente está preparado para trabalhar com aquela propriedade. Já a operação *undefineProperty* torna indefinida a propriedade.
- A operação *setProperty* define um valor para a propriedade identificada pela chave denominada *propertyName*. Se uma propriedade com este nome ainda não estiver definida, uma nova propriedade é criada e assinalada com o valor passado no parâmetro.
- A operação *addValue* adiciona o valor correspondente ao parâmetro *value* à propriedade identificada. Desta forma, assume-se que propriedades podem ter valores como sendo lista de valores.
- A operação *copyValues* copia os valores de uma propriedade para outra com o objetivo de guardar valores temporários de propriedades durante um processo de renegociação; a operação *removeValue* retira o valor indicado do conjunto de valores da propriedade. Já a operação *selectValue* seleciona um valor (indicado no parâmetro *value*) para a referida propriedade.

Conforme discutido, mudanças de valores de propriedades originadas pela aplicação requerem um cuidado especial. Trocas de valores de propriedades podem afetar a aplicação e o próprio *middleware*, uma vez que podem implicar em novos requisitos relacionados aos recursos da plataforma; e mais ainda: podem alterar inclusive o comportamento de outros componentes da aplicação. Diante disto, o *framework* designou para o *Configurator* a tarefa de verificar a viabilidade da mudança de valores de propriedades e a decisão de eventualmente mudar. Devido ao fato do *Configurator* ser o elemento que comanda todas as fases do ciclo de vida do sistema, ele sabe o momento conveniente para mudar a configuração. Neste contexto, definiu-se a interface *IUserSets* para controlar estas operações no escopo do *Configurator*.

O *Configurator* usa o *ApplicationProxy* para manter e gerenciar as operações de manipulação de propriedades. Uma definição ou alteração de valor de propriedade somente se concretiza após a verificação da consistência, por parte do *Configurator*; cabe ao *ApplicationProxy* apenas atualizar as mudanças nos componentes que tiverem valores de propriedades alterados. Para fazer esta atualização, o *ApplicationProxy* utiliza a interface *IPropertyUpdate* de cada componente envolvido, sinalizando aos respectivos recursos virtuais as mudanças de estado no ciclo de vida através da operação *changeStateTo*. A partir deste momento, cada componente pode realizar a sua alteração comportamental observando os novos valores de propriedade configurados.

A Fig. 4.18 descreve a interface *IUserSets* do componente *Configurator*. As operações que compõem esta interface são de fato implementadas pelo *ApplicationProxy*; este nível de indirecionamento foi definido para preservar a consistência da configuração da aplicação, ou seja, apenas as operações que deixem o sistema em estado coerente com a plataforma são repassadas para o *ApplicationProxy* correspondente. A cada operação desta interface, existe uma operação com mesmo nome, assinatura e finalidade na interface *IConfigurationSet*s do componente *ApplicationProxy*. De forma semelhante, algumas operações da interface *IStructuralSets* poderiam ser incorporadas pela interface *IUserSets*, o que permitiria suportar adaptações estruturais no modelo oriundas de interação dinâmica da aplicação; no entanto, considerando que os impactos desta permissão poderiam levar a situações imprevistas, decidiu-se que este tipo de adaptação requer uma análise mais ampla.

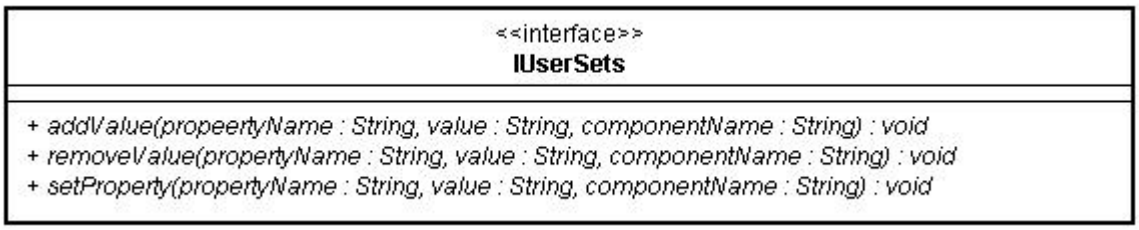


Fig. 4.18: Interface *IUserSets*

Para definir e ajustar configurações no sistema, muitas vezes é necessário realizar negociações complexas; um processo de negociação geralmente intercala entre ações de restrição e expansão nos valores de determinadas propriedades. Desta forma, para assegurar a recuperação de configurações anteriores, os seguintes conceitos podem ser explorados para este fim, utilizando as operações da interface *IConfigurationSets* do *ApplicationProxy:*

- **capacitação de uma propriedade do componente**: indica, em relação à propriedade, o universo de possibilidades de valores (capacidades) suportados;
- **valores nativos**: denotam para uma instância do componente, o conjunto de possíveis valores de uma determinada propriedade; ou seja, o conjunto de valores nativos deve ser necessariamente um subconjunto da capacitação;
- **valores preferidos:** correspondem a uma faixa que indica um subconjunto resultante das preferências definidas pela aplicação em relação a uma determinada propriedade do componente;
- **valor selecionado**: valor escolhido para a propriedade indicada em determinado componente.

Estes conceitos foram definidos originalmente no *framework* PREMO [7]; na abordagem do corrente trabalho há no entanto uma diferença em relação ao conceito de valor selecionado. Em nossa abordagem, um valor selecionado denota uma única instância de valor; já em PREMO, o conceito incorpora uma faixa de valores. Nosso entendimento é de que o valor selecionado corresponde à propriedade atual relacionada com a operação do sistema, a qual rege o comportamento do correspondente componente associado. Uma descrição visual destes conceitos é apresentada na Fig. 4.19.

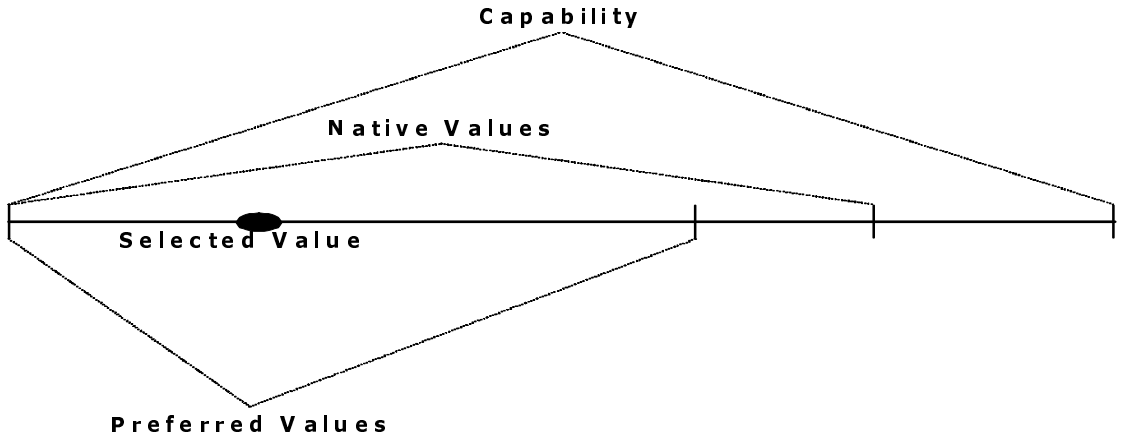


Fig. 4.19**:** Faixas de valores de propriedades

Com o objetivo de prover capacidade de introspecção e navegação na hierarquia de metacomponentes, tanto para fins de navegação por parte do *Configurator*, quanto para adaptação, a ser realizada pela própria aplicação, definiu-se a interface *IIntrospection*, apresentada na Fig. 4.20.

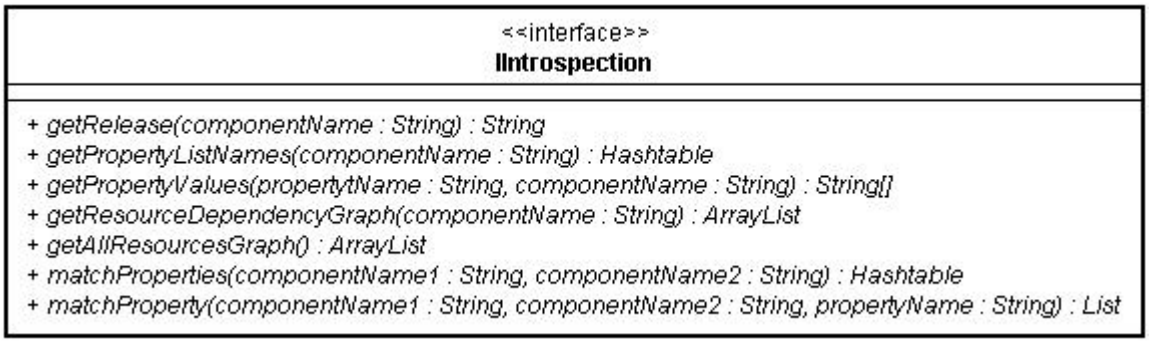


Fig. 4.20: Interface *IIntrospection*

A operação *getRelease* fornece a data de distribuição do componente, cujo nome é fornecido como parâmetro. A forma adotada para identificar componentes foi definir um modelo estruturado de

nomes, consistindo de quatro campos: nome do componente, nome da instância, nome da aplicação e o endereço IP do *host* onde ele reside.

De modo a prover facilidades para a realização de negociações envolvendo propriedades dos componentes, a interface *IIntrospection* define operações similares às apresentadas na interface *IPropertyInquiry* (Fig. 4.4) para obter, por exemplo a lista de chaves(nomes) de propriedades e valores de uma determinada propriedade. Adicionalmente, foram definidas duas operações úteis para o estabelecimento de casamentos envolvendo valores comuns de propriedades (*matchings*). Assim, caso se queira descobrir entre todas as propriedades de dois componentes quais são as comuns, utiliza-se a operação *matchProperties*; por outro lado, caso se queira obter, entre as propriedades comuns, quais são os valores presentes simultaneamente nas duas, utiliza-se a operação *matchProperty* indicando o nome da propriedade e os nomes dos respectivos componentes.

Para dar subsídios à realização de reconfiguração envolvendo mudanças dinâmicas na estrutura da aplicação, foram definidas as operações *getResourceDependencyGraph e getAllResourcesGraph*. A primeira retorna uma lista com os componentes, dos quais, o componente indicado no parâmetro depende, e a segunda operação, *getAllResourcesGraph*, retorna a lista com todo o grafo de dependência.

4.2.3 Suporte para QoS

Na presente seção são discutidas e apresentadas as interfaces do Cosmos relacionadas com os requisitos de QoS enumerados na Seção 2.3.3.

Para gerenciar os requisitos de QoS, o Cosmos define componentes arquiteturais ativos para verificar dinamicamente se os parâmetros de QoS do sistema se comportam de acordo com os requisitos da aplicação. Face à grande flutuação dos fatores associados à aplicação, plataforma e infra-estrutura de comunicação, o modelo utiliza o conceito de intervalos de QoS, no qual a definição dos requisitos ocorre por faixas de valores. Se durante a execução da aplicação for detectada alguma situação de não cumprimento da QoS negociada, um processo de adaptação é iniciado observando os requisitos estabelecidos na descrição da aplicação. Os componentes que fornecem as funcionalidades para provisão de QoS foram introduzidos na Fig. 4.14, no contexto da visão geral do modelo funcional do Cosmos. Adiante, a Fig. 4.21 apresenta uma visão destes componentes focada na descrição funcional do modelo de QoS, onde são abstraídas as interfaces que não estão relacionadas com a QoS. Os parágrafos seguintes discutem o papel de cada elemento do modelo destacado na figura.

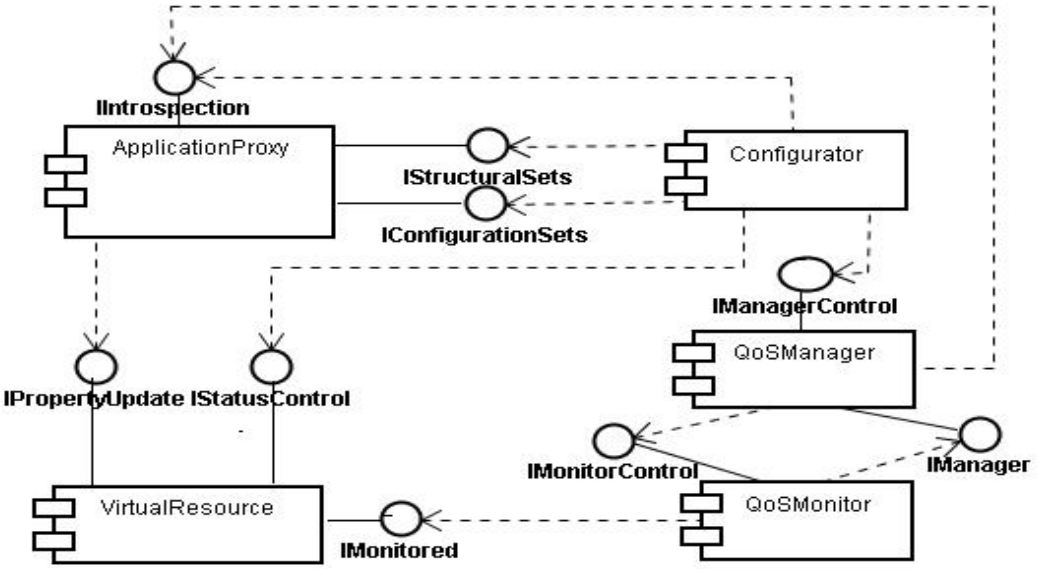


Fig. 4.21: Visão geral do modelo de QoS

Os gerentes de QoS (componentes *QoSManager*) são responsáveis pela instanciação e gerenciamento de monitores (*QoSMonitor*). Quando algum requisito de QoS estiver incoerente em relação aos parâmetros especificados pela aplicação, um monitor designado para observar este parâmetro notifica o fato ao respectivo gerente através de um mecanismo de chamada do tipo *callback*. O gerente associado deve notificar o *Configurator* para realizar um processo de adaptação. *QoSManagers* são informados pelo *Configurator* sobre o resultado do processo de adaptação através da operação *changeStateTo* da interface *IStatusControl* (descrita na Fig. 4.6). O *Configurator* também usa a interface *ISatusControl* de cada recurso virtual para disparar transições de estados, informando-os assim sobre o novo estágio dos mesmos no ciclo de vida.

Para comandar o processo de gerenciamento, o *Configurator* utiliza a interface *IManagerControl* do componente *QoSManager*. A Fig. 4.22 descreve esta interface, consistindo nas seguintes operações:

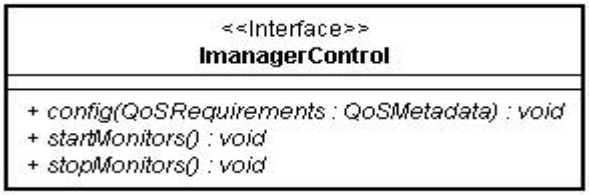


Fig. 4.22: Interface *IManagerControl*

- ***config***: Configura o *QoSManager* de acordo com os metadados passados como parâmetro. Estes metadados determinam os parâmetros que devem ser monitorados. O modelo de metacomponentes que foi descrito na Fig. 4.9 mostra apenas associações de metadados de QoS com conexões virtuais.
- ***startMonitors***: ativa os monitores instanciados para iniciarem a monitoração dos parâmetros de QoS especificados.
- ***stopMonitors***: suspende temporariamente a monitoração dos parâmetros de QoS especificados até a concretização da adaptação.

A interface *IManager*, representada na Fig. 4.23, apresenta a operação *callback* do componente *QoSManager*. Esta operação é usada por monitores para informar ao gerente que um parâmetro do recurso monitorado está fora da faixa de valores de QoS especificada. Os parâmetros da operação informam o valor obtido na observação (*value*) e a identificação do monitor (*tag*).

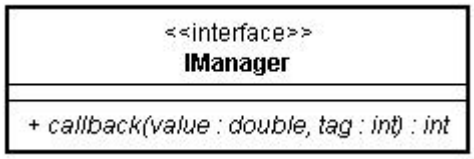


Fig. 4.23: Interface *IManager*

Um monitor de recursos, representado pelo componente *QoSMonitor*, obtém os dados do recurso virtual usando a interface *IMonitored* do mesmo (Fig. 4.24). Esta interface define as seguintes operações para o recurso virtual monitorado retornar informações de gerenciamento de QoS:

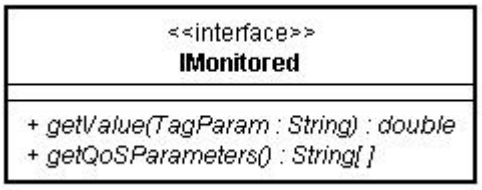


Fig. 4.24: Interface *IMonitored*

- ***getValue***: retorna o valor de QoS do recurso monitorado;
- ***getQoSParameters***: retorna uma lista com os possíveis parâmetros de QoS monitoráveis pelo *VirtualResource*.

Após a instanciação de um gerente de QoS, o *Configurator* define os metadados que descrevem os parâmetros de QoS a serem observados, cujos valores correspondem aos parâmetros contidos na especificação da aplicação. Para gerenciar cada parâmetro de QoS definido, o *QoSManager* cria um monitor para observar o recurso associado. Na criação de um monitor, o *QoSManager* responsável define os metadados relacionados com os requisitos de QoS do recurso que o monitor observa (cada monitor observa apenas um parâmetro de um recurso), bem como a freqüência da observação.

Ao ser notificado sobre uma violação de QoS, o gerente analisa e processa a notificação consultando e atualizando os metacomponentes do *ApplicationProxy*. Feita a análise, o *QoSManager* repassa a notificação ao *Configurator*, que coordena o processo de adaptação.

Para controlar e configurar monitores, *QoSManagers* utilizam a interface *IMonitorControl* (Fig. 4.25), que define as seguintes operações:

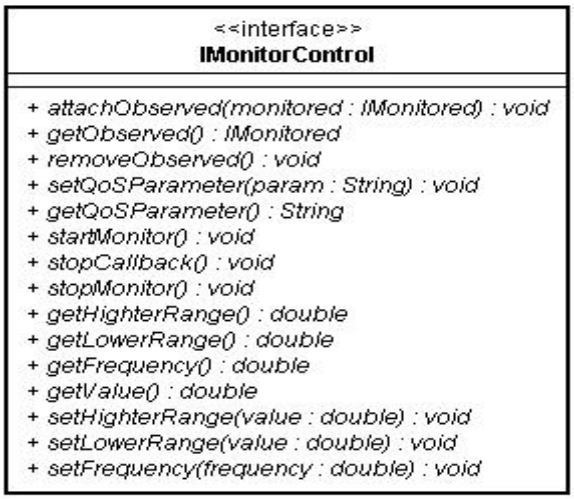


Fig. 4.25: Interface *IMonitorControl*

- *attachObserved*: indica ao monitor o recurso a ser monitorado (interface *IMonitored*);
- *getObserved*: retorna a referência para o recurso monitorado (interface *IMonitored*);
- *removeObserved*: remove a referência para o recurso monitorado (interface *IMonitored*);
- *setQoSParameter*: indica ao monitor qual parâmetro de QoS ele deve monitorar;
- *getQoSParameter*: retorna qual parâmetro de QoS está sendo monitorado;
- *startMonitor*: inicia o processo de monitoramento;
- *stopCallback*: suspende o mecanismo de *callback* (o mecanismo de *callback* é usado para informar o gerente que o recurso está operando numa faixa de QoS diferente da negociada);

- *stopMonitor*: suspende o processo de monitoramento;
- *getHighestValueOfRange*: retorna o limite superior da faixa de observação;
- *getLowestValueOfRange*: retorna o limite inferior da faixa de observação;
- *getFrequency*: retorna a freqüência de observação;
- *getValue*: retorna o valor atual do parâmetro de QoS monitorado;
- *setHighestValueOfRange*: configura o limite superior da faixa de observação;
- *setLowestValueOfRange*: configura o limite inferior da faixa de observação;
- *setFrequency*: configura a freqüência de observação.

A Fig. 4.26 apresenta um diagrama de seqüência UML com uma descrição do processo de instanciação de gerentes e monitores de QoS..

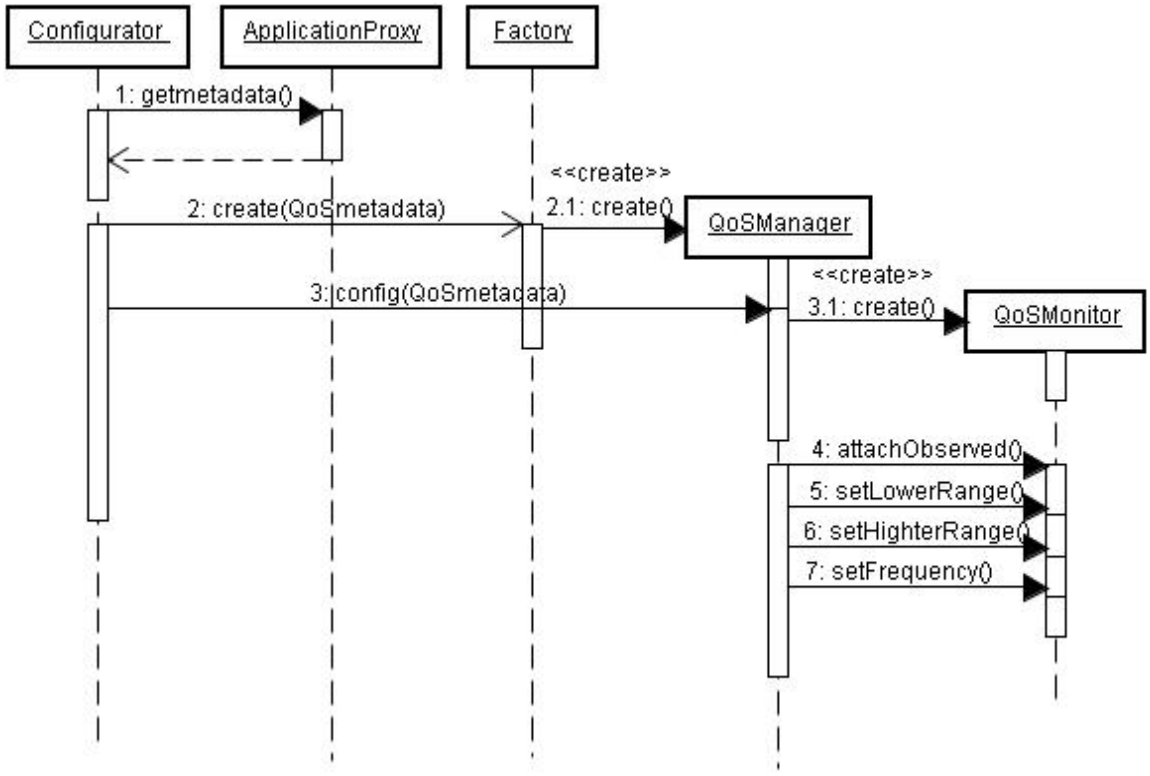


Fig. 4.26: Instanciação de gerentes e monitores de QoS

O processo começa com o *Configurator* solicitando os metadados relacionados aos respectivos parâmetros de QoS descritos no componente *ApplicationProxy*. Em seguida, o componente *Configurator* cria um gerenciador de QoS (*QoSManager*). Após o processo de instanciação do gerente, este é configurado de acordo com os metadados extraídos da especificação da aplicação, mediante processo de negociação. Um gerenciador é responsável por um grupo de componentes relacionados,

como por exemplo, os componentes envolvidos numa conexão virtual. Naturalmente, diferentes conexões terão diferentes gerenciadores. Em seqüência, o *QoSManager* cria os monitores e indica para os mesmos os recursos que serão monitorados (*attachObserved*), suas referências para fins de chamadas *callback*, bem como os parâmetros de QoS e as respectivas faixas que cada um deverá observar.

De acordo com o modelo apresentado para descrição de aplicações (Seção 4.1.4), o modelo de suporte à QoS define mecanismos para especificar e identificar regiões de QoS. A Fig. 4.26 apresenta uma ilustração com as principais operações definidas pelo Cosmos relacionadas com a configuração da QoS. As regiões têm suas faixas de valores de QoS associadas através de chamadas às operações *SetLowerRange* e *setHighRange*. Embora não estejam descritos na figura, valores *default* para propriedades podem ser definidos, servindo como referências para dar início à operação dos recursos nas respectivas regiões. As faixas associadas às regiões podem ser modificadas dinamicamente em função das políticas de gerenciamento especificadas pela aplicação. A definição envolvendo a seleção da região de QoS adotada também pode mudar dinamicamente, por exemplo, quando um monitor observar uma violação em relação à faixa associada à região corrente. Neste caso, o gerente de QoS, em conjunto com o *Configurator*, escolhem uma nova região, iniciando a operação nesta região com o respectivo valor de propriedade *default* definido para esta região. O modelo também permite a retomada automática para regiões de QoS melhores, a partir de tentativas a serem realizadas periodicamente, onde a periodicidade definida para realizar estas tentativas é especificada na descrição da aplicação. Após serem instanciados e configurados, os monitores aguardam o início da operação do recurso.

A Fig. 4.27 descreve um cenário envolvendo as operações relacionadas com o processo de adaptação na QoS. Ao dar início à execução da aplicação, o *Configurator* ordena o gerente (*QoSManager*) para iniciar a operação dos monitores através de uma chamada do método *startMonitors*. Por sua vez, o *QoSManger* repassa as ordens para cada um dos monitores associados através de chamadas à operação *startMonitor*. O gerente de QoS aguarda notificações de seus monitores, indicando possíveis violações de QoS (chamadas *callback*). Como parâmetros de uma chamada *callback* são passados ao correspondente gerente, o identificador do monitor (*tag*) e o respectivo valor detectado. Assim, conhecendo estas informações, o *QoSManager* seleciona dentre as possíveis regiões descritas na especificação, qual delas deverá ser a nova região escolhida para o recurso operar, indicando-a em seus metadados. Cabe ao *Configurator*, mediante consulta aos

metadados do gerente, avaliar a viabilidade e decidir pela realização desta adaptação. O *Configurator*, por sua vez, informa ao gerente o resultado do processo de configuração através da operação *changeStateTo* na interface *IStatusControl*. Enquanto aguarda a notificação, o gerente continua a operar normalmente para os demais parâmetros de QoS monitorados.

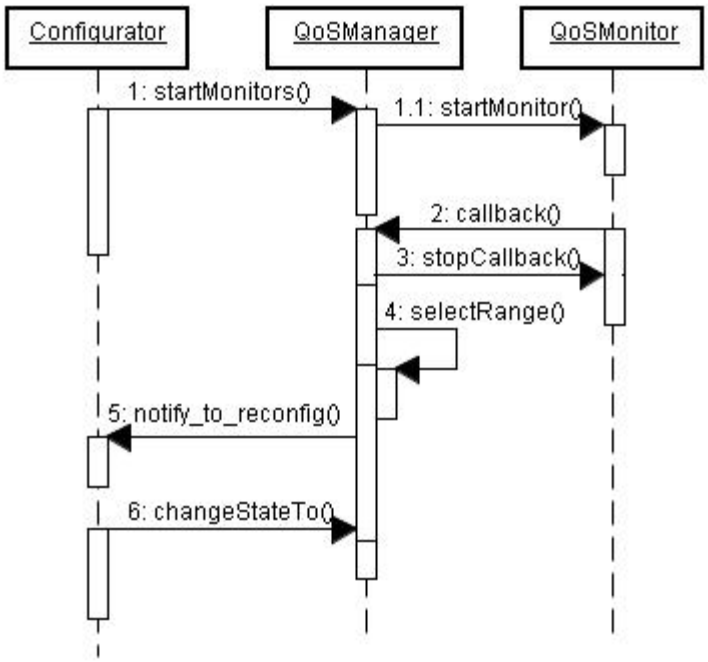


Fig. 4.27**:** Processo de Adaptação

A adaptação pode envolver diversos tipos de ações, como por exemplo: alteração na faixa de valores associada à região; alteração da freqüência de observação e mudança de QoS para execução em outra região de operação. Após o *QoSManager* receber do *Configurator* o estado resultante da adaptação (operação *changeStateTo*), o gerente de QoS pode eventualmente notificar o configurador para ele realizar uma nova adaptação, selecionando uma nova região a ser tentada.

Podem ocorrer situações em que um único monitor não seja capaz de verificar se um recurso está atendendo às expectativas de QoS, como por exemplo, na monitoração dos níveis de perda de pacotes numa rede. Neste caso, pode-se atribuir à conexão virtual uma responsabilidade adicional, de modo que ela também gerencie os parâmetros de QoS associados à rede usando uma hierarquia de componentes monitores. Desta forma, valores monitorados separadamente podem ser obtidos sistematicamente pela conexão virtual através de chamadas explícitas a métodos *getValue* das interfaces *IMonitorControl* dos monitores internos. Nesse caso, cabe à conexão virtual processar estes valores e disponibilizá-los em sua interface *IMonitored*. Assim, um monitor designado para observar se

este parâmetro de QoS da rede está, ou não, dentro da faixa de operação especificada, deve notificar o seu gerente respectivo, no caso de não cumprimento dos requisitos.

4.2.4 Suporte para Conectividade em Plataformas Distribuídas Heterogêneas

Esta seção apresenta e discute as funcionalidades dos componentes do modelo de interconexão definido no Cosmos, direcionando o foco para o suporte aos requisitos indicados na Seção 2.3.1. A Fig. 4.28 apresenta uma visão simplificada do modelo de interconexão com os componentes do modelo e os relacionamentos entre eles.

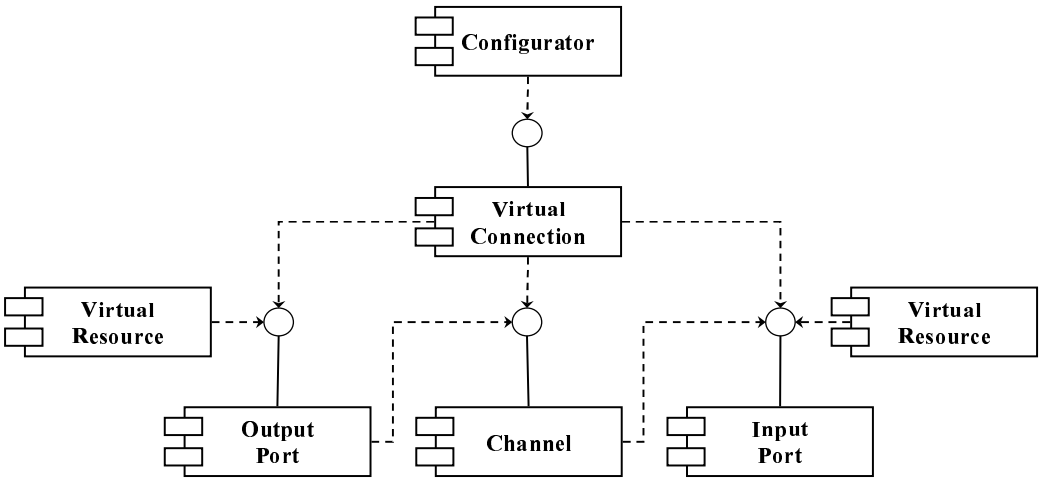


Fig. 4.28: Visão simplificada do modelo de interconexão de componentes.

No Cosmos, os componentes *VirtualResources* podem interagir com outros, como fonte ou destino de dados multimídia. Entretanto, para realizar uma interconexão entre componentes em sistemas distribuídos, muitas vezes é necessário construir um *pipe* envolvendo vários elementos internos, como filtros, redes, *switches* e sistemas operacionais.

De forma a reduzir a complexidade envolvida na conexão de componentes, o *framework* Cosmos explora os conceitos de portas e conexões virtuais, os quais provêem abstrações para tratar aspectos relacionados com formatos de mídia, topologias, sincronização, QoS e protocolos de comunicação.

No modelo de interconexão, portas (de entrada e saída) definem interfaces padrão, que são independentes dos tipos de tecnologia de comunicação suportadas nas plataformas onde os componentes serão instanciados.

O modelo introduz o conceito de canal de comunicação, cuja representação na arquitetura é suportada por um componente denominado *channel*. O objetivo deste componente é ocultar detalhes relacionados com tecnologias de comunicação que poderão vir a ser usadas para implementar conexões entre portas de componentes. Ou seja, os aspectos e especificidades de tecnologias envolvidas com o suporte à comunicação são encapsulados em canais, que se encarregam de prover interfaces padrões.

A separação explícita das portas e do respectivo canal de comunicação envolvidos numa conexão permite que plataformas aderentes ao modelo suportem diferentes abordagens e protocolos de comunicação. Assim, canais podem ser configurados para levar em consideração os tipos de dados do fluxo, os parâmetros de carga da rede e a localização dos componentes envolvidos (*VirturalResources*). Devido à flexibilidade do modelo, que permite vários tipos de ajustes, é necessário que plataformas aderentes assegurem a compatibilidade entre portas ligadas.

O modelo suporta várias topologias de interconexão, permitindo realizar conexões um-para-um, um-para-muitos, muitos-para-um e muitos-para-muitos. O modelo define dois tipos de direção de fluxo: entrada e saída. Assim, por exemplo, uma conexão entre uma porta de saída e várias portas de entrada constrói uma topologia do tipo um-para-muitos.

As portas são registradas em componentes *VirtualConnections*, os quais são responsáveis pelo gerenciamento das operações envolvidas. Um componente *VirtualConnection* representa uma conexão entre duas ou mais portas; a definição do número de portas (entrada e saída) depende das ligações descritas na especificação da aplicação. Portas se comunicam através de canais de comunicação. Um canal realiza uma ligação entre um par de portas, devendo assim existir um canal para cada par de portas.

O diagrama da Fig. 4.29 apresenta as interfaces funcionais relacionadas com o modelo de interconexão. No modelo, a configuração dos componentes envolvidos é realizada pelo componente *Configurator*. O *Configurator* é responsável por realizar acordos e negociações entre os componentes envolvidos de forma a cumprir os requisitos da aplicação. Para isto, ele deve verificar se há compatibilidade entre as capacidades da plataforma e os requisitos da aplicação.

Na Fig. 4.29 são mostradas duas interfaces operacionais usadas por recursos virtuais para enviar e receber dados, que são respectivamente, as interfaces *IInputPortOperation* e *IOutputPortOperation*. Assim, os recursos virtuais envolvidos numa conexão não precisam conhecer os tipos de comunicação suportados nas plataformas locais onde serão instanciados, nem tampouco controlar a operação da transmissão dos dados entre as respectivas portas que compõem o fluxo. O tratamento da sincronização

nos fluxos de dados é realizado, de forma transparente, pelas portas. As portas definidas no modelo adotam dois tipos de semânticas de operação: mensagens seqüenciais e fluxos contínuos.

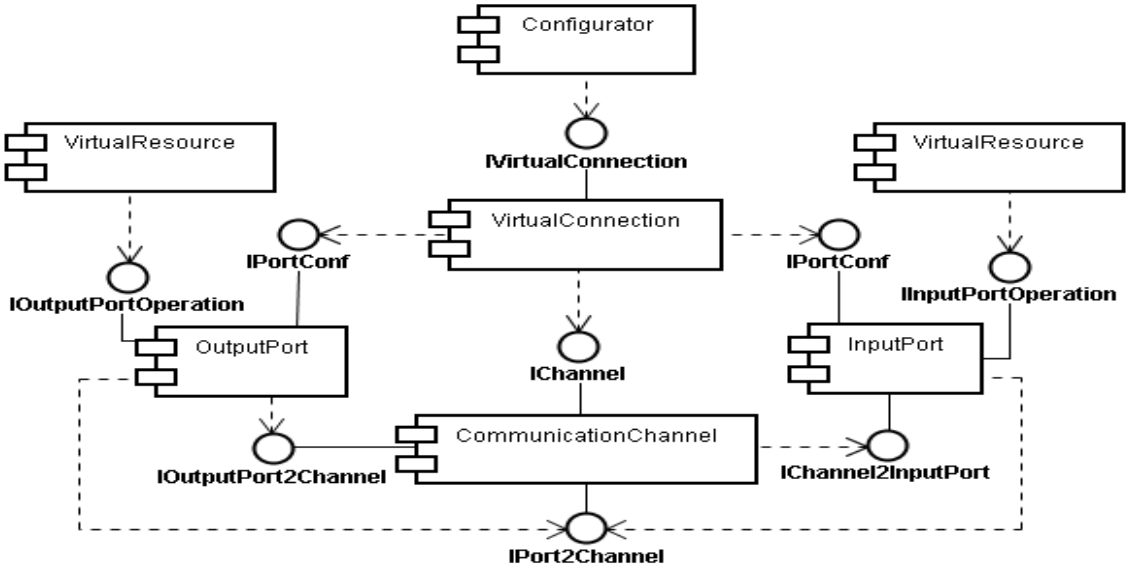


Fig. 4.29**:** Visão funcional do modelo de interconexão

Na abordagem orientada para mensagens seqüenciais, as portas de saída e entrada organizam o fluxo em uma seqüência de mensagens individuais. No lado da porta de entrada, o componente associado pode ler mensagens individuais num dado momento, preservando a seqüência correspondente à geração das mesmas. Já na abordagem orientada a fluxo, as portas de saída e entrada tratam o fluxo como uma seqüência de octetos individuais. No lado da porta de entrada, o componente pode obter todos os octetos disponíveis, sendo preservada também a ordem em que os mesmos foram gerados.

Adicionalmente, o modelo estabelece dois tipos de semânticas para interação entre componentes e portas: síncrona e assíncrona. Uma operação com semântica envolvendo uma chamada síncrona adota uma abordagem bloqueante, ao passo que a operação assíncrona adota uma semântica de chamada não-bloqueante. Assim, na semântica síncrona, um componente que chama a correspondente operação fica bloqueado até que a porta de entrada associada receba o dado. Contrariamente, na semântica assíncrona o componente não fica bloqueado, mesmo que, por exemplo, faça uma chamada de leitura em uma porta de entrada com *buffer* vazio usando uma operação associada (obviamente não receberia dado algum), ou tentasse enviar um dado numa porta de saída com *buffer* cheio.

A Fig. 4.30 descreve os métodos relacionados com as interfaces *IOutputPortOperation* e *IInputPortOperation* e as semânticas associadas com os métodos destas interfaces. Portas que provêem a interface *IOutputPortOperation* são caracterizadas como portas de saída. Elas oferecem métodos para envio de dados, usando tanto primitivas bloqueantes (operação *blockedSend*), como não-bloqueantes (operação *send*). Quando recursos virtuais usam a operação *send* não-bloqueante, eles precisam analisar o resultado retornado de maneira a decidir o que fazer, se por exemplo, não tiver espaço no *buffer* da porta associada. De forma similar, portas que provêem a interface *IInputPortOperation* são caracterizadas como portas de entrada. Como pode ser observado na Fig. 4.30, a interface *IInputPortOperation* define operações com semântica bloqueante (operações *waitMessage*, na abordagem orientada a mensagens e *waitBuffer,* na abordagem orientada a fluxos) e operações com semântica não-bloqueante (operações *getMessage,* na abordagem orientada a mensagens e *getBuffer,* na abordagem orientada a fluxo), de forma a dar suporte às duas abordagens semânticas de operação das portas.

A operação *reconfig*, definida na interface *IOutputPortOperation*, é utilizada por um componente emissor, no momento da adaptação, para notificar à porta de saída associada que, a partir deste momento, ela deverá enviar os dados do fluxo correspondente utilizando o novo canal.

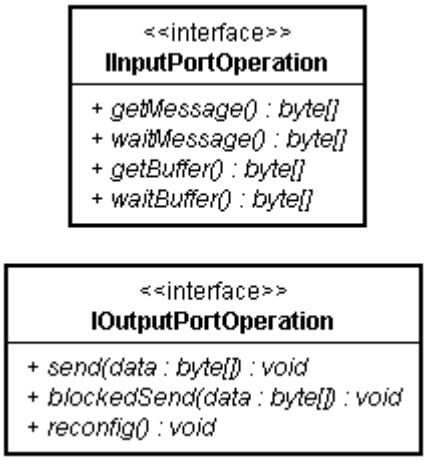


Fig. 4.30: Interfaces operacionais das portas

O *Configurator* usa a interface *IVirtualConnection* (Fig. 4. 31) para interagir com o componente *VirtualConnection*. Esta interface define os métodos usados para gerenciamento da conexão, incluindo métodos para adicionar portas (operações *attachInputPort* e *attachOutputPort*). Portas são identificadas no *Configurator* através de nomes simbólicos (*tags*) utilizados nos métodos para listar as portas associadas (operações *getInputPorts* e *getOutputPorts*), para recuperação de uma porta (operações

getInputPort e *getOutputPort*) e para remover uma porta de uma conexão (operações *removeInputPort* e *removeOutputPort*). O método *connect* é usado para iniciar um processo de conexão e o método *close* para encerrar e liberar os recursos associados. O método *reconfig* é usado pelo *Configurator* para iniciar um processo de adaptação. O método *getMetadata* é utilizado para fornecer as informações da conexão.

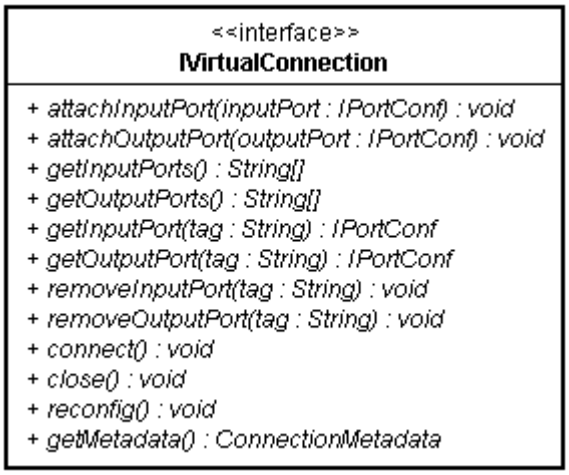


Fig. 4. 31: Interface *IVirtualConnection*

O componente *VirtualConnection* realiza as operações de configuração das portas utilizando a interface *IPortConf*. A Fig. 4.32 descreve as operações desta interface.

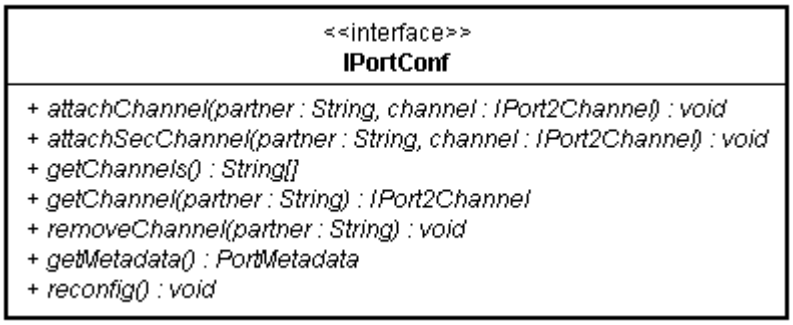


Fig. 4.32: Interface *IPortConf*

Uma porta pode ter vários canais associados. Portanto, é necessário que se forneçam operações para gerenciamento de associações entre portas e os respectivos canais. A interface *IPortConf* fornece o método *attachChannel* para adicionar um novo canal de comunicação à porta. A interface *IPortConf* também é usada para a realização de adaptação dinâmica através de associações de canais de comunicação secundários (operação *attachSecChannel*). Canais secundários são usados por uma porta

de saída para enviar dados em um novo formato durante um processo de adaptação. A interface também fornece métodos para recuperar a lista de canais associados (operação *getChannels*), ou apenas um canal (operação *getChannel*) e para remover um canal (operação *removeChannel*). O método *getMetadata* é utilizado para obter os metadados que descrevem a porta, e o método *reconfig*, para realizar ações de suporte ao ajuste das propriedades da porta, como por exemplo, criar um novo *buffer*.

A interface *IChannel* (Fig. 4.33) é usada pela conexão virtual para gerenciar canais de comunicação; ela oferece operações para: abrir o canal – ou seja, ativá-lo, colocando o mesmo em operação (operação *open*); para fechar, desativando a operação do mesmo (operação *close*); para associar uma porta de entrada aonde o canal deverá colocar os dados do fluxo associado (operação *attachTarget*); e para recuperar esta porta(operação *getTarget*). A operação *clone* é usada durante o processo de adaptação para duplicar um canal de comunicação, construindo assim um canal secundário. Um canal secundário deverá ser usado em um processo de adaptação para iniciar a transmissão de um novo fluxo de dados, cujo formato tenha sido previamente negociado. Ao final da fase de adaptação, o canal secundário se torna principal.

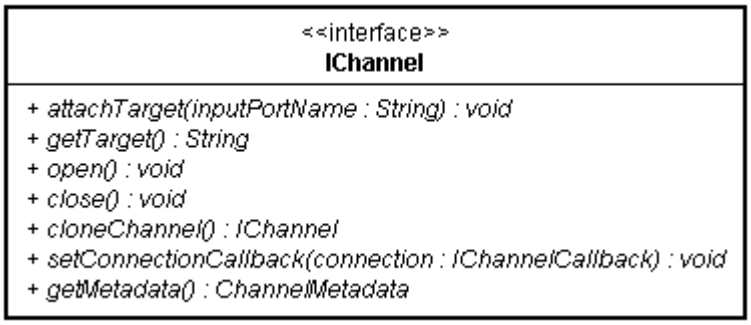


Fig. 4.33: Interface *IChannel*

IChannel define a operação *setConnectionCallback* para indicar uma interface de *callback* a ser utilizada após a adaptação, para informar à conexão virtual que um determinado canal não é mais necessário.

Canais de comunicação e suas portas associadas interagem entre si para trocarem dados usando as seguintes interfaces: *IPort2Channel*, *IOutputPort2Channel* e *IChannel2InputPort*. A Fig. 4.34 descreve estas interfaces.

A interface *IOutputPort2Channel* é usada por portas de saída para enviar dados através do canal. Na outra extremidade, o canal de comunicação usa a interface *IChannel2InputPort* para repassar os dados para a correspondente porta de entrada. O parâmetro *tag* é usado para determinar durante a

fase de adaptação se o canal em questão é o canal principal, ou se é um canal secundário. A interface *IPort2Channel* fornece o método *release*, que poderá ser usado pelas portas para indicar quando o canal associado não for mais necessário.

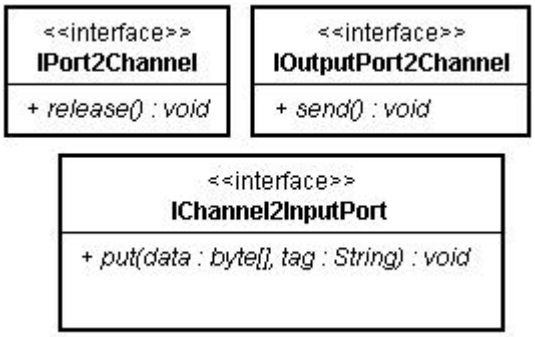


Fig. 4.34: Interfaces para comunicação entre canais e portas

A definição do número de canais em uma *VirtualConnection* depende da topologia de interconexão estabelecida na especificação da aplicação. Seguindo o modelo proposto, podemos ter tanto conexões *unicast* (cardinalidade 1:1 no relacionamento *VirtualConnection-Channel*), quanto *multicast* (cardinalidade 1:N no relacionamento *VirtualConnection-Channel*). Ou seja, para realizar uma conexão *multicast*, eventualmente o sistema criará diversos canais de comunicação, podendo assim envolver diversas tecnologias de comunicação simultaneamente.

É através da configuração de uma aplicação, onde conexões são estabelecidas, que uma determinada topologia de interconexão será definida. O processo de escolha de uma determinada tecnologia depende de fatores dinâmicos, tais como a localização dos componentes envolvidos (por exemplo, se estão instanciados no mesmo espaço de endereçamento ou em computadores distribuídos). O processo para conectar dois ou mais componentes consiste em verificar se existe compatibilidade entre os tipos de dados das portas, cujos formatos associados são descritos na especificação dos componentes, as capacidades de cada componente envolvido e a topologia requerida. Para isto ele estabelece valores de propriedades que satisfaçam os requisitos descritos na especificação.

Para a construção de uma aplicação, componentes precisam ser interconectados. A Fig. 4.35 descreve a fase de estabelecimento de uma conexão. No exemplo da figura, estão envolvidas questões de negociação de propriedades, criação de componentes e configuração. Para não sobrecarregar a figura, alguns métodos e parâmetros foram omitidos.

Durante o processamento da especificação, o *Configurator* solicita a criação das portas a um componente do tipo fábrica (*PortFactory*), conforme definido pelo Cosmos, e negocia as propriedades

com o objetivo de definir valores compatíveis entre os requisitos especificados e as capacidades da plataforma. Este processo produz os metadados para as operações que irão efetivamente realizar a conexão das portas.

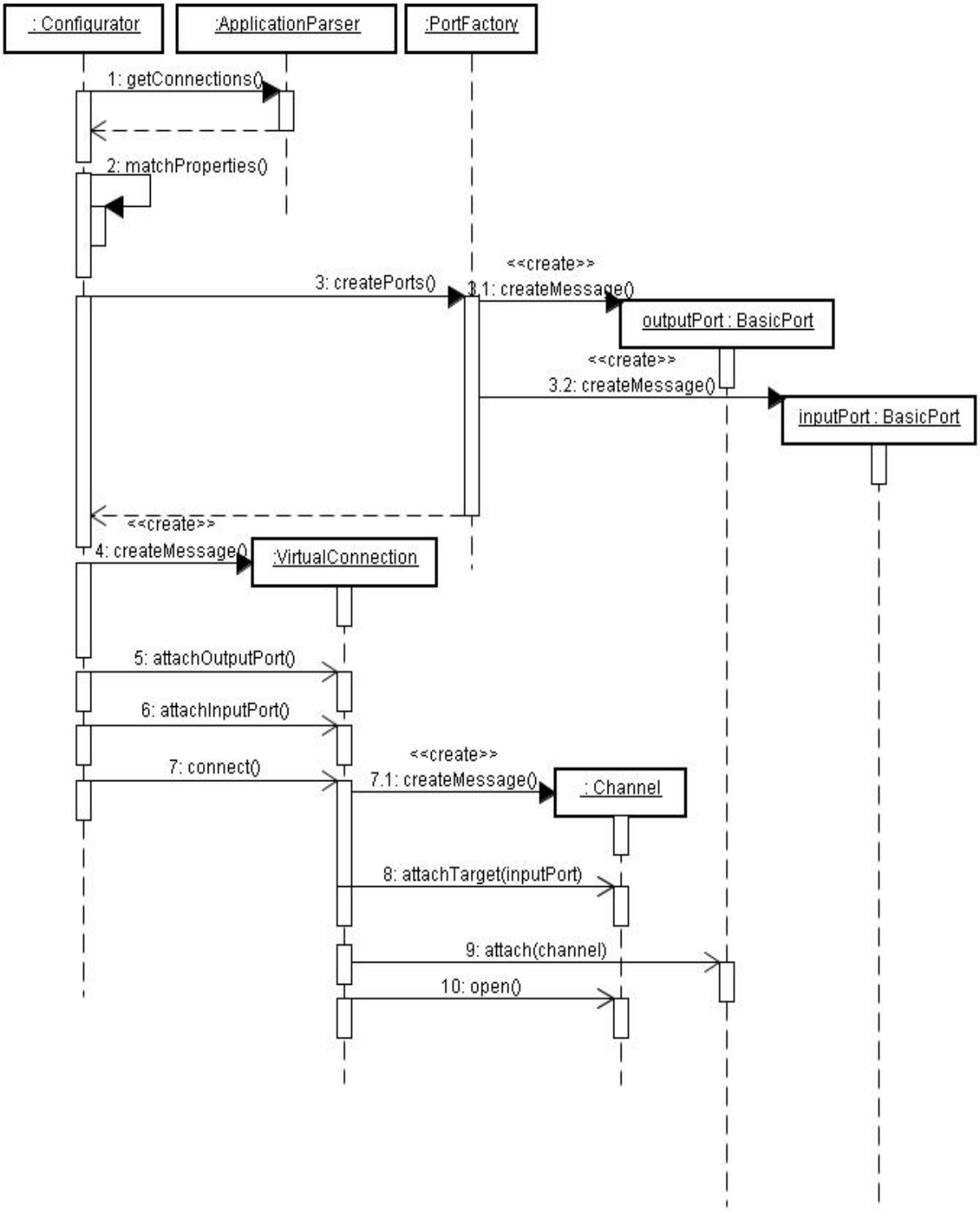


Fig. 4.35: Estabelecimento de uma conexão

Em seguida, o *Configurator* cria uma instância do componente *VirtualConnection* para representar a conexão em questão, configurando-a com os metadados referentes à conexão. As portas criadas anteriormente são anexadas ao *VirtualConnection*. As ações relacionadas aos registros das portas junto ao componente *VirtualConnection* correspondem à realização da ligação entre as mesmas. Após a configuração do componente *VirtualConnection,* o *Configurator* realiza uma chamada para o método *connect* que efetivamente dispara o processo de alocação dos canais de comunicação necessários.

O componente *VirtualConnection* realiza a instanciação dos canais de comunicação passando para cada um deles seus respectivos pares de portas. Em seqüência, ele passa para a porta de saída uma referência do canal e, a seguir, passa para o canal, uma referência para a porta de entrada correspondente.

O modelo fornece suporte para as políticas de adaptação reativa e pró-ativa definidas no *framework* Cosmos. Conforme mencionado anteriormente, na adaptação reativa a aplicação é quem indica as operações necessárias; em uma adaptação pró-ativa, cabe à aplicação apenas definir na especificação os requisitos e as políticas. Assim, o *middleware* define o tipo de adaptação considerando as informações passadas.

De maneira a facilitar a compreensão do processo de adaptação em conexões, a abordagem no texto está dividida em duas fases: uma fase de preparação, onde o *Configurator* notifica os componentes envolvidos indicando que um processo de adaptação foi iniciado, e uma fase de realização, onde os componentes efetivamente ativam as mudanças, alterando o seu comportamento. A Fig. 4.36 mostra um exemplo de um diagrama de seqüência com as ações correspondentes à fase de preparação de uma adaptação pró-ativa.

Uma adaptação pró-ativa é originada por solicitação de um componente gerente de QoS; o *Configurator* recebe uma sugestão de indicação de novos valores e valida esta indicação (operação *validate*), de modo a não deixar o sistema em um estado inconsistente. Para isso, o *Configurator* identifica os componentes que potencialmente podem ser afetados pela mudança, iniciando um novo processo de negociação para verificar se estes componentes suportam a alteração indicada. Caso a verificação resulte em sucesso, é iniciado então o processo de adaptação, onde o *Configurator* comunica os componentes envolvidos (recursos virtuais *sender* e *receiver*) para que se preparem para a reconfiguração através de uma chamada da operação *changeStateTo* (interface *IStatusControl*, descrita na Fig. 4.6), indicando *Reconfiguring* (estado descrito na Seção 4.3) como sendo o novo estado. O

Configurator solicita então à conexão virtual para realizar a tarefa de reconfiguração dos componentes associados (portas e canais), através da operação *reconfig* no diagrama da Fig. 4.36. Em seqüência, o componente *VirtualConnection* se prepara para criar um novo canal de comunicação e realizar a troca do antigo para o novo, com o sistema em operação. Para isto, *VirtualConnection* chama a operação *reconfig* nas portas envolvidas para que elas se prepararem, criando por exemplo, novos *buffers* para receber o novo fluxo de dados; em seguida, o canal de comunicação associado é clonado e configurado de maneira a dar suporte a comunicação entre as portas existentes considerando os novos requisitos. As portas envolvidas sabem como gerenciar o canal clonado de forma a garantir a sincronização. Quando as portas e o canal estiverem prontos para a troca, o componente *VirtualConnction* notifica o *Configurator* de forma que ele dê início à segunda parte do processo de adaptação.

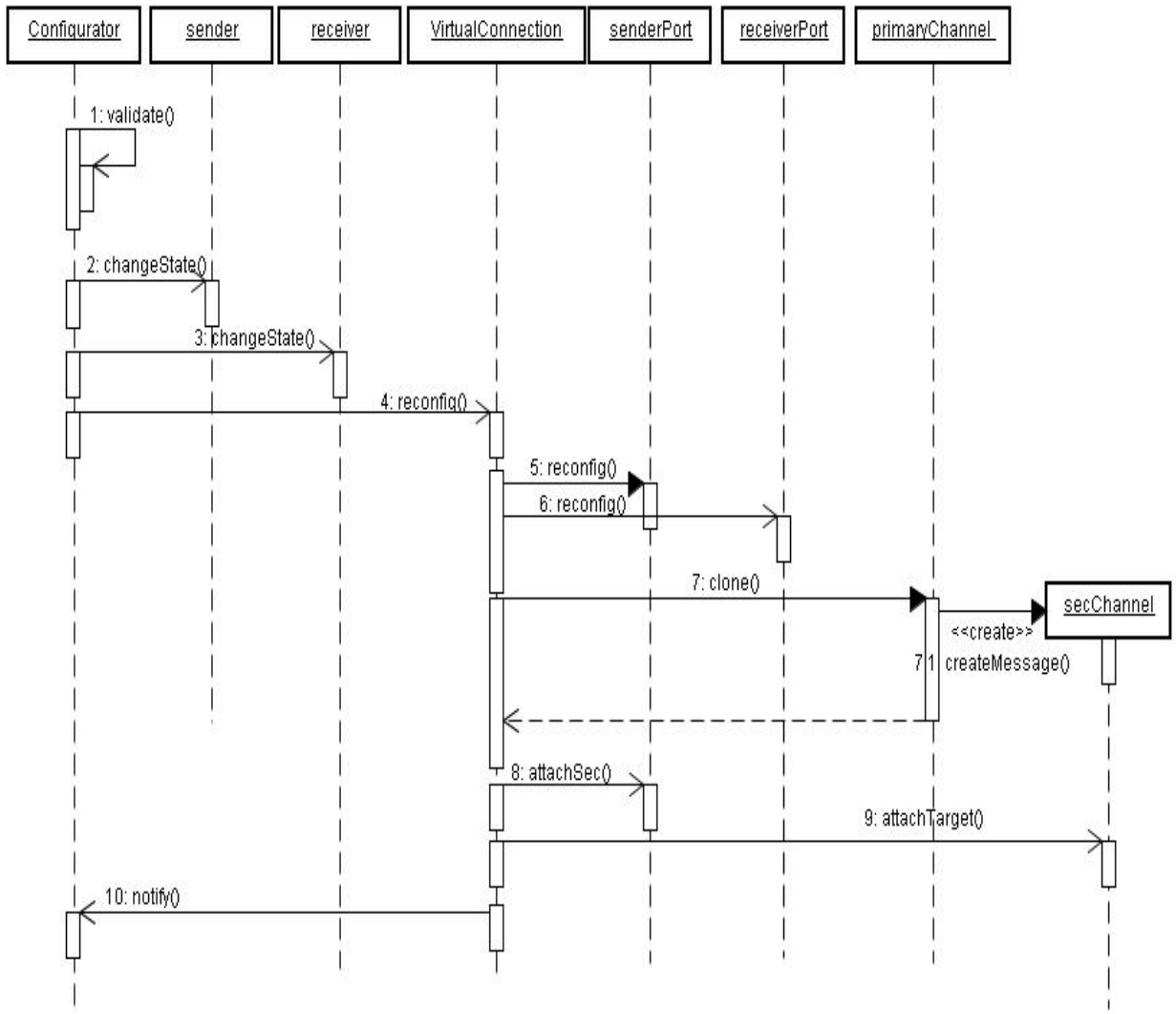


Fig. 4.36: Fase de preparação de uma adaptação pró-ativa

A segunda fase do processo de adaptação corresponde à ativação das mudanças. A Fig. 4.37 apresenta um exemplo com um diagrama de seqüência envolvendo as ações da fase de ativação do processo de adaptação.

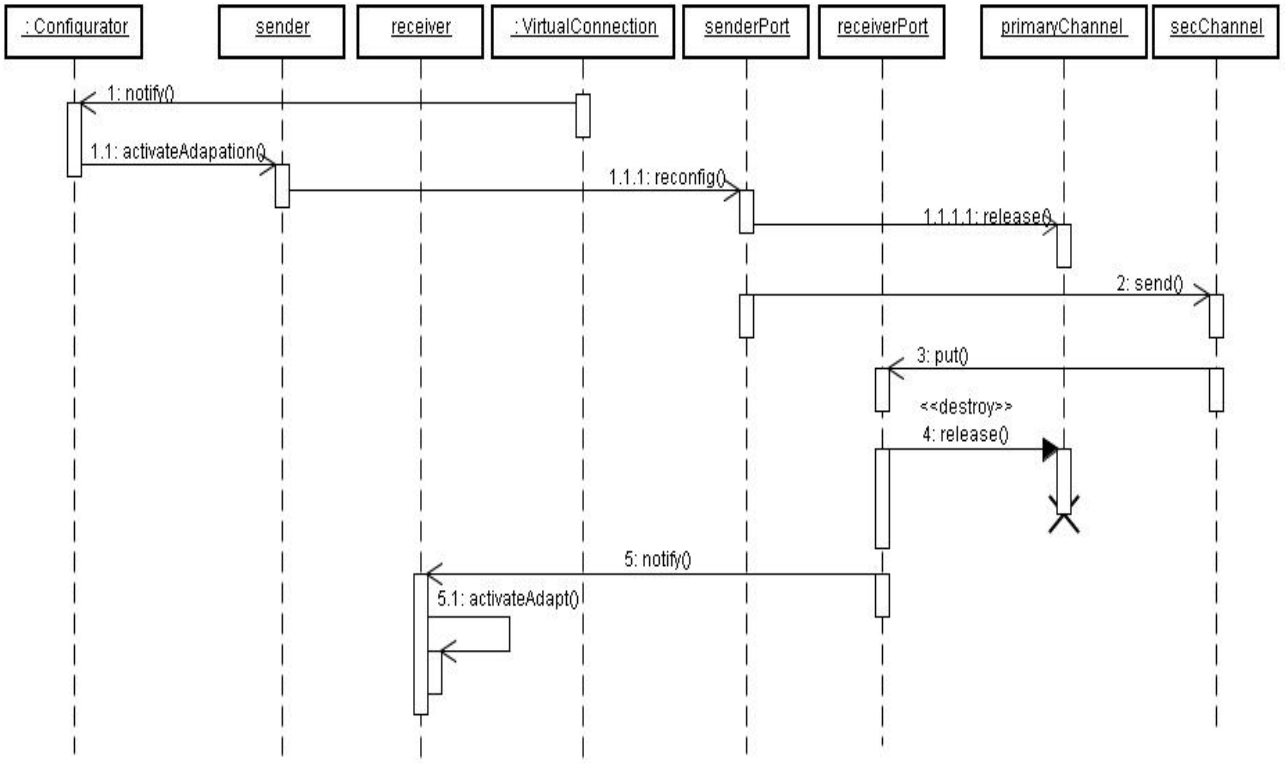


Fig. 4.37: Ativação de uma adaptação

Após a notificação ser enviada pelo componente *VirtualConnection*, indicando que o ambiente está pronto para mudança, o *Configurator* autoriza os componentes para mudarem as suas propriedades através da operação *activateAdaptation*. Esta operação representa, de forma simplificada, as ações do *Configurator*, correspondendo a mudança do estado e a mudança de valores de propriedades. No entanto, cabe ao componente transmissor esperar o momento em que a mudança do formato do fluxo possa ocorrer de forma consistente, como por exemplo, no término da transmissão de um quadro de vídeo. Desta forma, antes do iniciar a transmissão do quadro seguinte no novo formato, o transmissor indica a troca de formatos para a porta de saída associada através da operação *reconfig*. Esta chamada sinaliza a mudança, ou seja, a ação correspondente consiste em ativar o novo canal de comunicação. A porta indica então que não precisa mais do antigo canal, solicitando a sua liberação (operação *release*) e começa a enviar dados pelo novo canal alocado. Dados do novo fluxo são colocados em um novo *buffer* do canal secundário recentemente ativado. A porta de entrada espera até esvaziar o antigo *buffer*

notificando o correspondente recurso que a contém, quando ele estiver vazio; neste momento, o recurso *receiver* ativa a adaptação trocando as propriedades de forma a processar corretamente o novo fluxo de acordo com o novo formato. A porta de entrada então libera o antigo canal (operação *release*), indicando que ele não é mais necessário e muda a indicação para que o novo *buffer* seja utilizado na condição de execução em estado normal, ao passo de que o antigo *buffer* será desativado.

4.3 Construção de uma Aplicação e Gerenciamento de Recursos Associados

O modelo definido para o desenvolvimento de uma aplicação no Cosmos (Fig. 4.38) tem uma estrutura similar à proposta pelo ambiente Aster [156], consistindo nas seguintes fases:

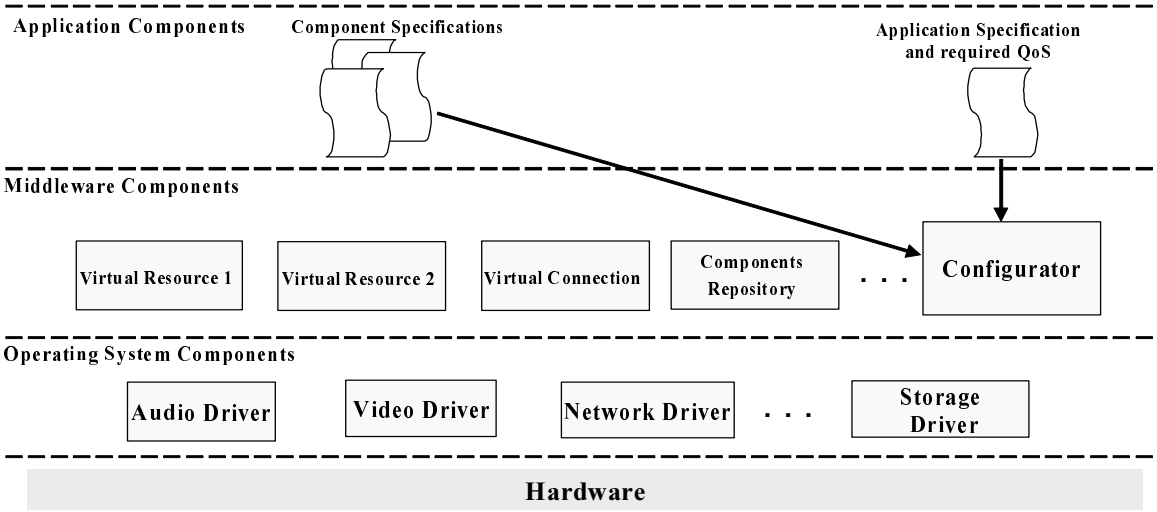


Fig. 4.38. Construção de uma Aplicação

- Fase de especificação da aplicação, onde a configuração dos componentes da aplicação é descrita de acordo com a linguagem de especificação disponível na plataforma *middleware*. Caso a plataforma *middleware* não tenha em seus serviços de repositório componentes capazes de atender os requisitos da aplicação, cabe à aplicação a tarefa de desenvolver novos componentes. Para isso, ela pode utilizar as facilidades de reuso incorporadas no *framework* Cosmos, estendendo as funcionalidades dos componentes básicos definidos originalmente. De maneira a serem reutilizados por outras aplicações, estes novos componentes precisam ser descritos na linguagem de especificação adotada pelo *middleware*.

- A segunda fase corresponde à Configuração do sistema. É nesta fase que o componente *Configurator* realiza as negociações de propriedades verificando se há compatibilidade entre a especificação, os tipos dos componentes envolvidos nas interconexões definidas e as capacidades da plataforma.
- A terceira etapa corresponde à fase de execução e monitoramento da QoS.

A arquitetura para gerenciamento de recursos definida no Cosmos envolve vários componentes. Considerando que o *Configurator* é o componente que tem a responsabilidade de coordenar todo o processo, o *middleware* precisa então instanciá-lo primeiramente. A partir dele, serão processados, configurados e executados os demais componentes do *middleware* e da aplicação.

Uma vez instanciado, o componente precisa ser configurado. Uma abordagem preliminar sobre a configuração de recursos virtuais no Cosmos foi apresentada em [31] [75]. Essa abordagem definiu um conjunto de etapas para o processo de configuração. Esse conjunto foi ampliado para contemplar a fase de adaptação dinâmica [83] [84], conforme mostrado na Fig. 4.39, consistindo nas seguintes etapas: negociação de propriedades suportadas, alocação de recursos requeridos, execução do componente, reconfiguração (denotando ajustes com o sistema parado) e adaptação (denotando ajustes com o sistema em execução).

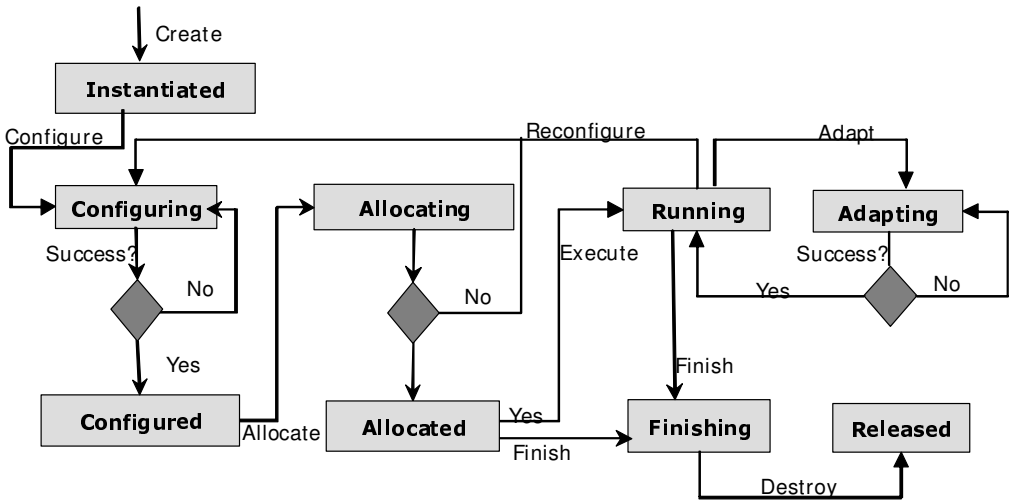


Fig. 4.39: Configuração de Recursos Virtuais.

De acordo com a arquitetura, os recursos requeridos associados à uma instância de um recurso virtual só serão efetivamente alocados quando todos os componentes descritos na especificação da

aplicação forem instanciados, as conexões correspondentes realizadas e a QoS negociada. Neste momento, a validação efetiva dos recursos necessários poderá se realizar de forma consistente, uma vez que todo o grafo da aplicação estará montado, fornecendo assim uma visão global dos recursos requeridos. Caso os componentes da aplicação tenham disponíveis os recursos que precisam, eles são alocados segundo um acordo previamente estabelecido pelo *Configurator*, cujas informações são mantidas no *ApplicationProxy*.

Analisando o diagrama de estados da Fig. 4.39, observa-se que o estado *instantiated* define a fase inicial do ciclo de vida, imediatamente após a criação de um componente *virtualResource*. Após serem criados, componentes são descritos em metacomponentes associados (atribuição de valores de propriedades), os quais são armazenados e gerenciados pelo respectivo componente *ApplicationProxy*. Ressalta-se que a transição do estado de *instantiated* para *configuring* corresponde à atribuição dos valores de propriedades nativas da instância.

O processo prossegue de forma similar, instanciando e negociando propriedades, até que todos os componentes da aplicação sejam instanciados. Uma vez que o *Configurator* tenha concluído o processo de negociação envolvendo os recursos locais e analisado a configuração de recursos remotos, ele solicita ao *ApplicationProxy,* através da operação *activate*, para ele indicar e repassar para cada componente da aplicação a informação de ativação do respectivo componente, ou seja, do início da sua execução.

A Fig. 4.40 mostra a arquitetura definida pelo *framework* Cosmos para dar suporte à execução de várias aplicações em um mesmo domínio. De acordo com a arquitetura, o *Configurator* cria um componente *ApplicationProxy* para cada aplicação. O exemplo da figura descreve uma situação envolvendo diferentes aplicações: A, B, até uma enésima. Durante o processo de configuração, o *Configurator* interage com o *ApplicationProxy* de cada aplicação para incluir, ou recuperar, informações ou propriedades de recursos virtuais, construindo os correspondentes grafos de metacomponentes associados. De acordo com o modelo definido no Cosmos, uma adaptação originada pela aplicação deve ocorrer através de operações na API do usuário (Interfaces *IUserConfiguration* e *IUserSets*). Assim, a eventual mudança na configuração precisa ser realizada sob controle do *Configurator*; cuja concretização ocorre através da atualização de propriedades nos metacomponentes do *ApplicationProxy.* Para modificar uma configuração, a aplicação pode realizar consultas sobre o estado atual da plataforma e da própria aplicação usando a API de reflexividade (interface *IIntroscpection*). Por sua vez, o *Configurator* pode tomar a iniciativa de realizar a adaptação acessando

diretamente as interfaces *IStrutucturalSets* e *IConfigurationSets* do componente *ApplicationProxy*. Esta última abordagem é chamada na tese como adaptação pró-ativa, onde o *middleware* realiza a adaptação segundo a orientação estabelecida na especificação da aplicação.

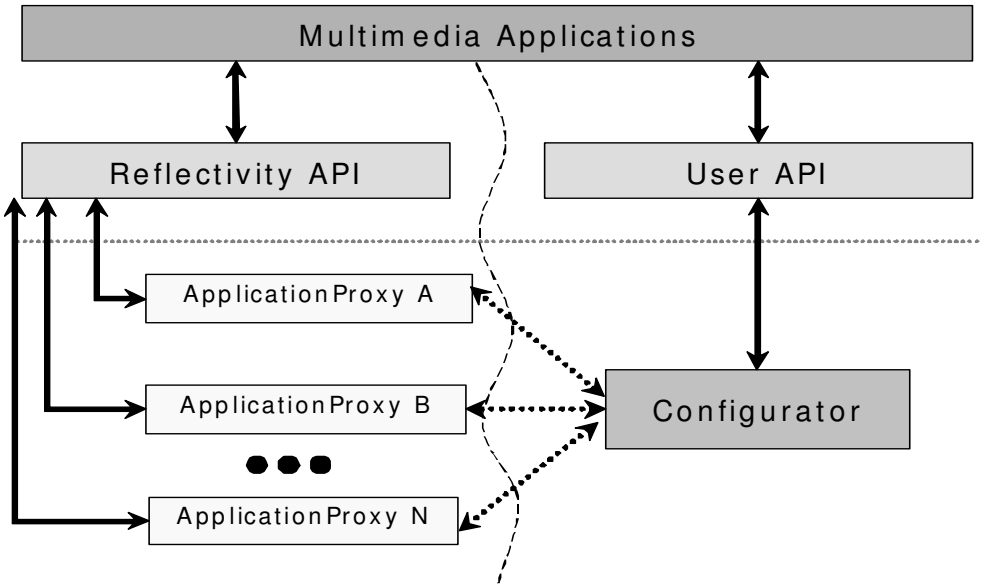


Fig. 4.40: Configuração de Aplicações

Durante a fase de execução, possíveis interações do usuário podem alterar os requisitos da aplicação ou do sistema; uma alteração no ambiente de execução de uma aplicação eventualmente pode provocar modificações no ambiente das outras aplicações. Uma situação exemplo é explorada e ilustrada no Capítulo 5.

Para atender aos requisitos de reconfiguração dinâmica, o *framework* provê os conceitos de reflexividade, cujos mecanismos de suporte são providos pelas interfaces de configuração definidas no contexto do componente *ApplicationProxy* e gerenciadas pelo *Configurator*. Uma adaptação que ocorre em resposta a uma ação direta da aplicação, através da API *IUserSets*, é denominada, conforme mencionado anteriormente, como adaptação reativa; operações desta interface têm acesso indireto às operações do *ApplicationProxy*. Na adaptação pró-ativa, o *middleware* usa diretamente a API de reflexividade.

Um processo de adaptação pró-ativa envolve o modelo de QoS definido pelo Cosmos, consistindo então na manipulação dos metadados associados. Estes metadados descrevem as possibilidades indicadas na especificação.

4.4 Considerações Finais

Este capítulo apresentou e discutiu os principais conceitos e características do *framework* Cosmos. A abordagem arquitetural do *framework* definido oferece níveis elevados de flexibilidade, provendo suporte a características de interoperabilidade, QoS e adaptabilidade. Como elementos chaves para suportar estas características, destacam-se os modelos de metacomponentes, QoS, interconexão e configuração.

A arquitetura do Cosmos provê um mecanismo de interconexão genérico, possibilitando a integração de uma variedade de tipos de componentes em ambientes heterogêneos e distribuídos como a *Internet* e os sistemas de TVDI. A API proposta para o modelo é relativamente simples e bastante flexível, cujas operações definidas dão suporte aos principais componentes definidos no *middleware* AdapTV.

Uma característica do Cosmos que não está presente nos modelos convencionais de Componentes é a abordagem para gerenciamento de QoS. No modelo, o tratamento de QoS se dá por faixas de valores de QoS, o que permite flexibilizar os níveis de tolerância a variações em função do tipo da aplicação e do estado da plataforma.

Utilizando o *framework* proposto, o capítulo seguinte descreve um protótipo de uma plataforma de *middlware* baseada no Cosmos, que foi implementado usando os conceitos e componentes básicos definidos. Dentre os componentes definidos no *framework* utilizados no desenvolvimento do protótipo, destacam-se como principais os modelos para representação e gerenciamento de recursos, propriedades, configuração, QoS, interconexão e adaptação dinâmica.

O modelo de interconexão constitui-se numa importante contribuição, pois, com a convergência das várias mídias e tecnologias associadas, como por exemplo a *Internet* e a TVDI, novas aplicações poderão ser construídas explorando estilos arquiteturais do tipo *pipes* envolvendo elementos intermediadores. Neste contexto, o componente *VirtualConection*, associado com as facilidades de configuração dinâmica do Cosmos, provê níveis elevados de flexibilidade. O conceito clássico de portas, que tem sido amplamente utilizado pela engenharia de *software*, permite tratar conexões *unicast* e *multicast* com abstrações similares.

No escopo deste trabalho, as questões relacionadas com o modelo de eventos não foram discutidas. As análises realizadas no contexto do projeto concluíram que os mecanismos definidos na linguagem Java eram suficientes para dar suporte ao desenvolvimento da plataforma protótipo do

middleware AdapTV, descrito no próximo capítulo. No entanto, devido as especificidades das classes de aplicações, um módulo de eventos deverá ser objeto de trabalhos futuros.

Capítulo 5

O *Middleware* AdapTV

O presente capítulo descreve o *middleware* AdapTV – um *middleware* para Sistemas de Televisão Digital Interativa definido como prova de conceito para a arquitetura proposta para o Cosmos. Para avaliar a viabilidade de implementação deste *middleware*, um protótipo foi implementado considerando os aspectos de configuração e gerenciamento de adaptação em uma aplicação simples, onde foram tratados requisitos de QoS no contexto de uma plataforma distribuída.

Conforme a discussão realizada na Seção 3.4, sistemas de TVDI requerem infra-estruturas adaptativas e suporte à configuração e reconfiguração dinâmicas. Neste cenário, o resultado da experiência desta implementação é uma contribuição da tese para o processo de desenvolvimento dessa nova tecnologia no cenário brasileiro atual. Antes de apresentar o *middleware* definido, o texto inicia mostrando o papel do *middleware* nos Sistemas de Televisão Digital Interativa .

5.1 Sistemas de Televisão Digital Interativa

Atualmente, Sistemas de Televisão Digital Interativa (TVDI), de forma semelhante aos sistemas multimídia tradicionais, envolvem uma diversidade de requisitos e características. Em adição, existe uma heterogeneidade de plataformas e vários tipos de dispositivos, recursos de *hardware* e *software*, bem como usuários com diferentes necessidades. Conseqüentemente, para lidar com esta diversidade de situações, sistemas TVDI devem prover uma infra-estrutura altamente adaptativa que dê suporte a requisitos como: confiabilidade, segurança, portabilidade, sincronização, extensibilidade e reflexividade [88].

Para o desenvolvimento de sistemas de televisão interativa, deve-se também levar em consideração o nível de interatividade que a infra-estrutura de transmissão proverá aos seus usuários. Este nível de interatividade pode ser classificado em duas categorias: sistemas sem canal de retorno e sistemas com canal de retorno [76].

A Fig. 5.1 apresenta o papel do *middleware* com uma visão simplificada de um sistema de televisão digital com canal de retorno. O cenário descrito contempla dois tipos básicos de entidades: provedores de serviços e clientes usuários. No cliente (dispositivo de exibição), o *middleware* lida com a especificação, recuperação, gerenciamento, apresentação e execução de fluxos de dados multimídia e componentes de *software*. Para exercer estas responsabilidades, a plataforma deve utilizar um sistema operacional com características de tempo real (RTOS). No provedor (estação produtora-transmissora), fluxos (*streams*) elementares de vídeo, áudio e dados são multiplexados para compor um único fluxo de transporte. Este fluxo é transmitido em um canal de *broadcast* codificado no formato MPEG2-TS [53]. Para prover características de interatividade em sistemas de TVDI sem canal de retorno, um mecanismo denominado Carrossel pode ser utilizado para injetar periodicamente programas aplicativos e dados no fluxo MPEG. Estes elementos são utilizados pelo *middleware*, por exemplo, para prover um pseudo-nível de interatividade, com opções de escolha limitada aos programas transmitidos juntamente com o fluxo. Mesmo em situações onde existe canais de retorno, o carrossel pode ser utilizado para transmitir os programas e dados selecionados pelas aplicações.

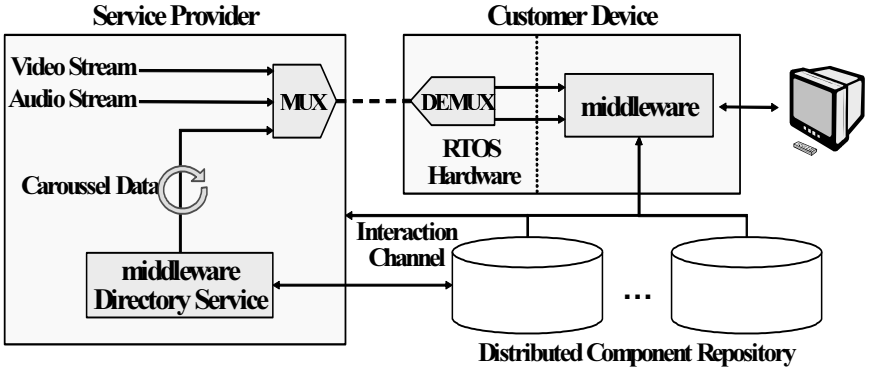


Fig. 5.1: Estrutura de um sistema de televisão interativa.

5.2 Arquitetura do *Middleware*

Para tratar as questões relacionadas com sistemas TVDI, conforme foi mencionado na seção anterior, um sistema de suporte deve prover uma infra-estrutura altamente adaptativa. Vários requisitos destes sistemas correspondem aos enumerados na Seção 2.3. Neste sentido, o sistema AdapTV apresentado neste capítulo foi definido usando os conceitos e componentes básicos do *framework* Cosmos. A implementação do AdapTV propicia a oportunidade de avaliar o Cosmos no contexto de uma tecnologia emergente.

Nesse contexto, a definição e implementação de um protótipo do *middleware* AdapTV e o desenvolvimento de uma aplicação-exemplo servem como prova de conceito da viabilidade e potencialidade do Cosmos para o cenário de sistemas multimídia distribuídos.

A arquitetura do AdapTV foi estruturada em camadas [31] [77]. A próxima seção apresenta uma visão estrutural das camadas do *middleware* e, na seqüência, o texto descreve os componentes da arquitetura explorados no protótipo.

5.2.1 Arquitetura em Camadas

A arquitetura do *middleware* AdapTV consiste das seguintes camadas: camada de configuração e gerenciamento e camada de manipulação de mídia. Estas camadas provêem APIs especializadas para a camada de aplicações, conforme observado na Fig. 5.2.

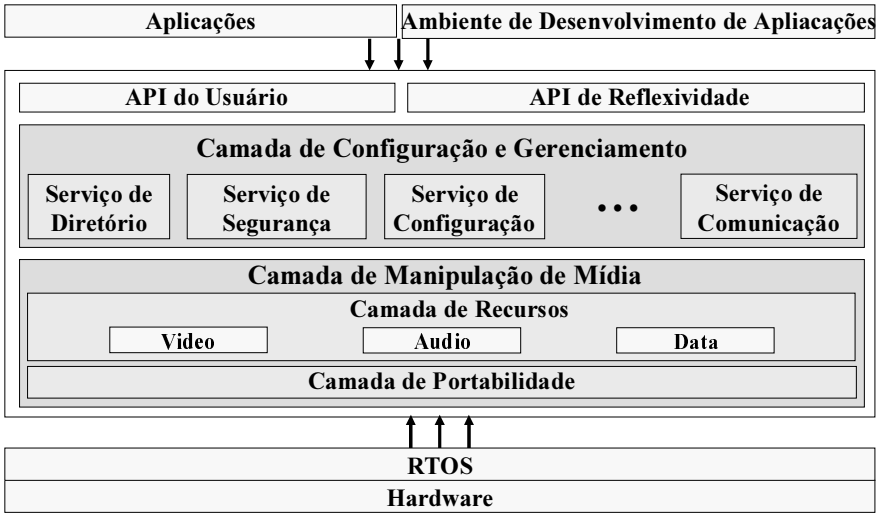


Fig. 5.2: Arquitetura do AdapTV em camadas

Na camada superior do *middleware*, encontram-se duas interfaces de programação (APIs): usuário e reflexividade. Estas APIs fornecem a base para a construção de ambientes de desenvolvimento e de aplicações interativas. O uso destas APIs permite acessar os recursos de *hardware* e do sistema operacional de maneira transparente às suas especificidades.

A API do usuário provê um conjunto de operações básicas que fornecem as funcionalidades tradicionais do *middleware*, como por exemplo, acesso à rede, seleção de programas, e localização de componentes na rede. Por outro lado, a API de reflexividade disponibiliza um conjunto de

operações para obtenção de informações sobre o ambiente. Através desta API, as aplicações e o próprio *middleware* podem modificar a configuração do ambiente de acordo com as duas formas de adaptação mencionadas anteriormente:

- **Reativa:** aplicações usam a API de reflexividade para obter metadados dos componentes e eventualmente para realizar adaptações estruturais.
- **Pró-ativa**: componentes de monitoramento do *middleware* reconhecem a necessidade de adaptação da aplicação, por exemplo quando um valor inadequado de QoS for atingido, de forma a requisitar ao *middleware* para realizar a adaptação.

A camada de configuração e gerenciamento engloba várias funcionalidades, incorporadas em componentes típicos, como os indicados a seguir:

- **Serviço de Configuração**: responsável por configurar, negociar, alocar e gerenciar componentes e recursos virtuais;
- **Serviço de Diretório**: um exemplo de serviço de repositório do Cosmos para realizar a recuperação de componentes requeridos de forma transparente;
- **Serviço de Segurança**: provê mecanismos para autenticação e controle de acesso;
- **Serviço de Transações**: provê suporte transacional para aplicações de comércio eletrônico;
- **Serviço de Comunicação**: suporta o desenvolvimento de aplicações distribuídas;

A camada de configuração e gerenciamento não tem acesso direto aos dispositivos e aos recursos do sistema operacional. Acessos a estes componentes são feitos através da camada de manipulação de mídia, onde são definidos recursos virtuais especializados para tratar cada tipo básico de fluxo (vídeo, áudio e dados), bem como para os demais aspectos de baixo-nível da plataforma.

Devido à existência de diferentes sistemas operacionais e à variedade de tipos de dispositivos, parte da implementação de um dispositivo virtual torna-se específica para cada plataforma ou dispositivo. Assim, estas partes de implementação apresentam baixo nível de portabilidade. Para amenizar o impacto desta restrição decidiu-se subdividir a camada de manipulação de mídia em duas subcamadas:

- **Camada de Recursos**: consiste dos recursos virtuais que definem APIs uniformes para os componentes de níveis mais altos, ao passo que assume a existência de APIs uniformes para acessar a camada de portabilidade;

- **Camada de Portabilidade**: composta de componentes que definem uma API uniforme para os componentes específicos da camada de recursos; estes componentes isolam os aspectos de implementação específicos, incorporando a parte de código dependente de cada tipo de plataforma e de dispositivo.

Seguindo o modelo de componentes do Cosmos, os componentes do AdapTV possuem interfaces funcionais e interfaces para manipulação de metadados, as quais são usadas para definição das suas propriedades (por exemplo, parâmetros de QoS, tipos de formatos suportados, valores de propriedades preferidos).

A arquitetura em camadas proposta para o AdaptTV [31] [77] é apresentada de forma detalhada na Fig. 5.3:

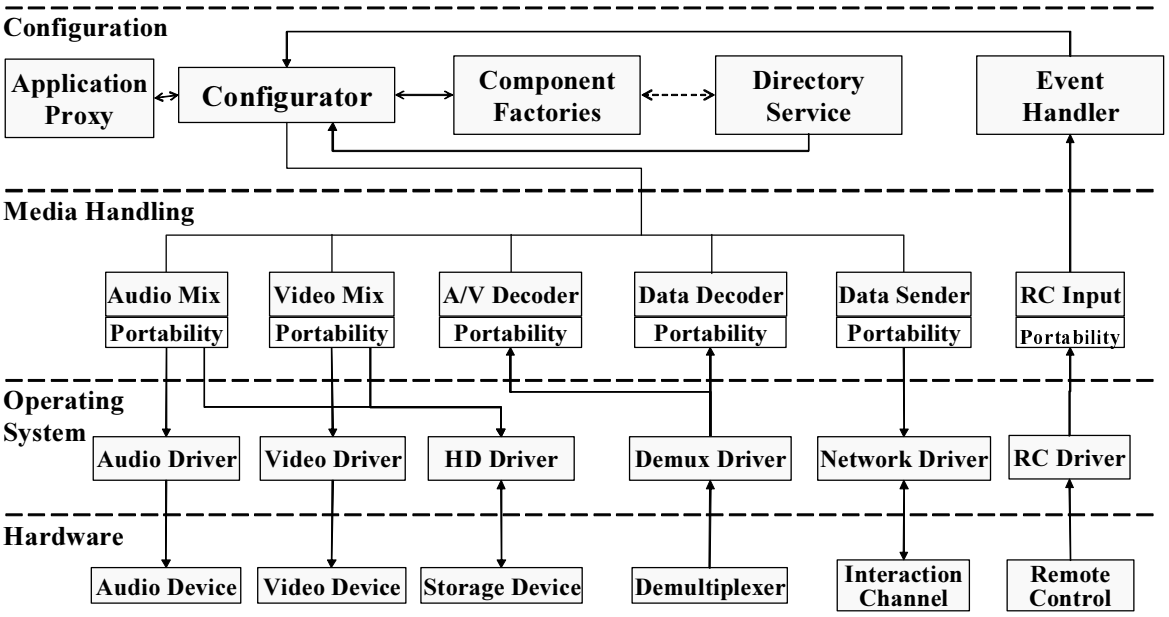


Fig. 5.3: Arquitetura Detalhada do AdapTV

A implementação realizada do AdapTV se limita a alguns componentes da camada de configuração desta arquitetura. Pode-se observar na Fig. 5.3 que a definição dos componentes desta camada corresponde aos componentes básicos do modelo arquitetural apresentados na Seção 4.1.3. Apesar do gerenciador de eventos não ser discutido no Capítulo 4, foi mencionada a sua importância para o modelo de configuração. No presente protótipo, onde a aplicação desenvolvida é simples, foram exploradas as facilidades da linguagem Java para processamento de eventos. Outra simplificação do

protótipo está relacionada com o serviço de diretório. Na implementação, cada *Configurator* mantém um repositório com os componentes que ele pode instanciar e gerenciar.

Dentre os principais componentes considerados na implementação do protótipo, destacam-se os seguintes: *Configurator*, *ApplicationProxy*, *XML-Parser*, *VirtualResource*, *VirtualConnection*, *InputPort*, *OutputPort*, *QoSManager*, *QoSMonitor* e *Factories*. Apesar do componente *Parser* oferecer suporte para responsabilidades intrínsecas do *Configurator*, ele é tratado de forma independente do Cosmos, de modo que o *middleware* e a aplicação possam ter liberdade para definir sua própria linguagem. No protótipo foi definido um *parser* para uma linguagem baseada em XML.

Como a presente implementação foi desenvolvida no nível de prova de conceito, não foi estabelecido como propósito cobrir integralmente todas as funcionalidades do Cosmos; especificamente, o esforço foi concentrado nos modelos de metacomponentes, interconexão e QoS associados. No protótipo não foram implementados serviços de diretórios, conexões *multicast* nem adaptações envolvendo troca de componentes. As funcionalidades típicas de sistemas de TVDI se limitaram à transmissão e apresentação de vídeo com adaptação dinâmica baseada em mecanismos de QoS.

5.2.2 Linguagem de Especificação

Como discutido nas seções iniciais, Cosmos requer uma especificação das aplicações. No entanto, o *framework* não estabelece uma linguagem específica. No protótipo do AdapTV, definiu-se um modelo de especificação baseado em XML para descrever aplicações e componentes. Este modelo foi descrito inicialmente em [77] e ajustado posteriormente para incorporar os modelos de metacomponentes (descrito na Seção 4.1.4), de QoS (descrito na Seção 4.2.3), e de interconexão (descrito na Seção 4.2.4). Apesar desta implementação usar uma linguagem específica, novos *parsers* poderão ser incorporados no futuro, um vez que o modelo de metacomponentes proposto e a arquitetura de implementação definida para o AdapTV oferecem níveis de flexibilidade que permitem a troca do *parser*.

A decisão de adotar XML deve-se ao fato de existir uma grande quantidade de manipuladores e ferramentas XML, além de ser uma tecnologia adotada e experimentada nos padrões atuais mais importantes para televisão digital, como MHP [135], DASE [136] e ARIB[138].

5.2.2.1 Modelo de Especificação de Aplicações

Em conformidade com o modelo de especificação descrito no Cosmos, para o AdapTV uma aplicação é especificada e tratada explorando o conceito de composição. Esta seção apresenta as principais características de uma linguagem de especificação definida para descrição dos metadados das aplicações e dos componentes do AdapTV com base na sintaxe XML, assim como as estratégias de composição suportadas pelo *middleware*. A apresentação da linguagem é feita através de um modelo UML com uma estrutura similar à descrição do modelo RAS [150]. Esta abordagem tem como objetivo facilitar a apresentação, tornando mais clara a definição dos elementos sintáticos dos *XML Schemas* associados. Explorando essas facilidades de descrição, ambientes baseados no *middleware* podem criar facilmente novas aplicações e componentes. Para dar suporte a estas facilidades, o *Configurator* utiliza um *parser*, conforme o modelo introduzido no esquema de processamento de uma especificação apresentado na Fig. 4.10.

A Fig. 5.4 apresenta um diagrama UML com a descrição dos elementos correspondentes aos *XML-Schemas* definidos para a linguagem de especificação de aplicações para o AdapTV considerando os modelos apresentados nas figuras Fig. 4.1 e Fig. 4.9.

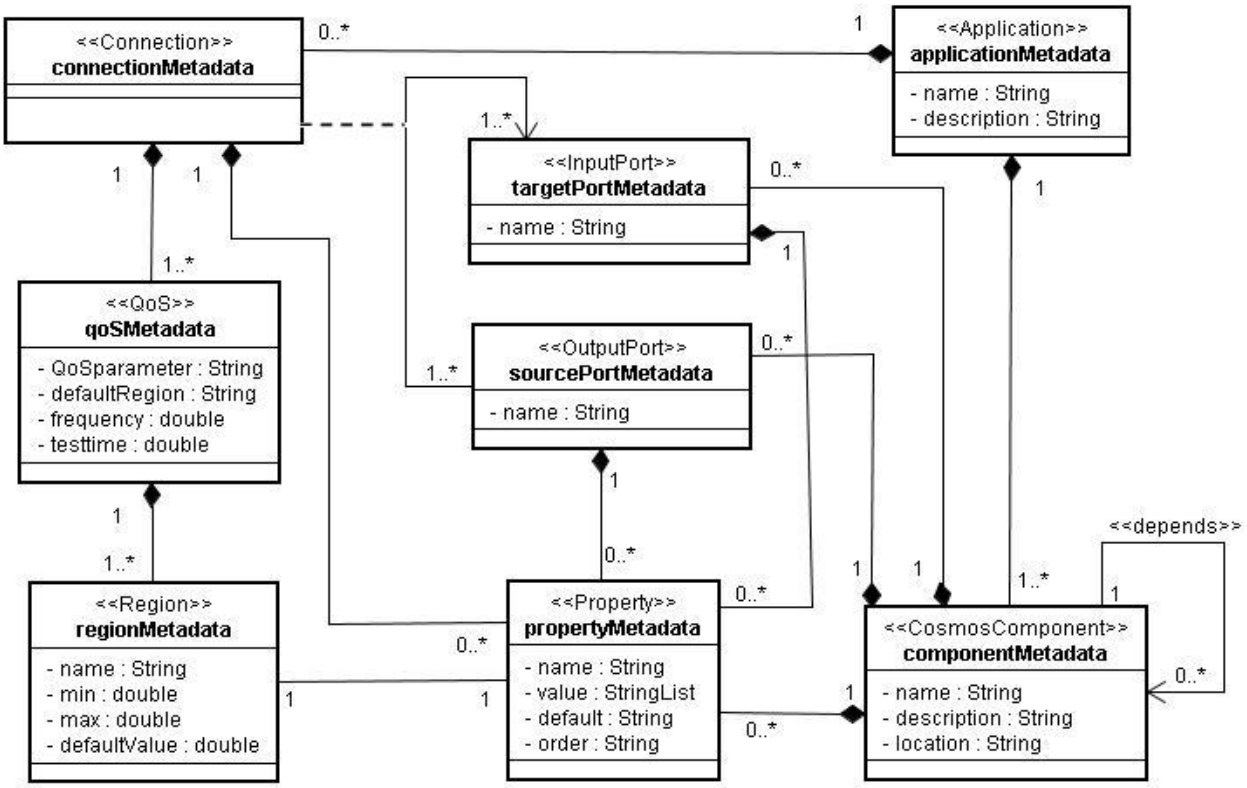


Fig. 5.4: Modelo UML para Descrição de Aplicações no AdapTV

No AdapTV, uma aplicação é descrita por uma composição de metadados estruturados de acordo com o modelo da Fig. 5.4. A aplicação (elemento que denota o nó raiz da hierarquia XML) é identificada na especificação por um nome e tem uma descrição associada, devendo ser composta por, no mínimo, um componente (restrição indicada na cardinalidade da associação entre *applicationMetadata* e *ComponentMetadata*). O modelo de descrição também permite a definição de composições de componentes indicada na figura através do relacionamento *depends*.

Os metadados associados à descrição de componentes de uma especificação correspondem ao nome, descrição e localização dos mesmos. Além destes, a descrição pode definir:

- **DEPENDÊNCIAS**: relações que identificam os componentes dos quais o componente descrito depende. Conforme mostrado na Fig. 5.4, dependências são indicadas através da associação entre os componentes *componentMetaData*.
- **PROPRIEDADES*:*** características enumeradas em *propertyMetadata*, normalmente utilizadas para descrever requisitos não funcionais;
- **RESTRIÇÕES:** elementos enumerados em *constrainMetadata* e utilizados para descrever eventuais restrições nos valores das propriedades dos componentes associados; o metacomponente *constraintMetadata* não está ilustrado na Fig. 5.4 porque ele têm a mesma estrutura do componente *propertyMetadata*;
- **PORTAS**: elementos usados para identificar as portas de entrada (*targetPortMetadata*) e as portas de saída do componente descrito (*sourcePortMetadata*).

Uma propriedade define um conjunto de valores para os quais aquela propriedade poderá ser instanciada. A ordem de preferência (usada por exemplo, para expressar o nível de QoS que interessa à aplicação) dos elementos deste conjunto de valores é definida de acordo com o atributo *order*, que pode assumir os valores a*sc* ou *desc*, indicando ao *Configurator* que, no processo de escolha, dê prioridade aos elementos do início ou fim deste conjunto, respectivamente.

Portas são identificadas por cadeias de caracteres (*name*), tendo listas de propriedades, restrições e atributos associadas. Propriedades identificam características da porta, como por exemplo, tamanho do *buffer* e taxa de transmissão.

A descrição dos metadados de uma conexão (elemento *connectionMetadata*) envolve a descrição das portas de saída (*sourcePortMetadata*) e entrada (*targetPortMetadata*) relacionadas, dos requisitos de QoS e estratégias de adaptação e das propriedades associadas à conexão.

Os principais elementos envolvidos com a descrição dos metadados de QoS são os seguintes:

- ***QoSparameter***: atributo que identifica um parâmetro de QoS monitorado.
- ***defaultRegion***: atributo usado para especificar a região de QoS escolhida para iniciar a operação do sistema, sendo definida como padrão para o recurso monitorado.
- ***frequency***: atributo usado para definir a freqüência de observação da QoS.
- ***testtime***: atributo utilizado para especificar, em milisegundos, a periodicidade em que o componente *Configurator* deve tentar mudar da região atual de QoS para outra qualitativamente próxima.

Os metadados que descrevem regiões de QoS definem o nome da região e um *range* que especifica os limites superior e inferior, determinando a região associada.

A estrutura, conteúdo e semântica dos principais elementos da linguagem e os *XML Schemas* definidos em [77] [83] [84] são apresentados de forma detalhada no Apêndice B.1.

5.2.3 Arquitetura de Implementação

A arquitetura de implementação definida para o AdapTV considera o modelo de metacomponentes descrito na Seção 4.1.2 do Capítulo 4 e explora as definições da linguagem baseada em XML apresentadas na seção anterior.

No que diz respeito ao modelo para implementação de componentes, a arquitetura segue a tendência da maioria das abordagens atuais ao adotar a tecnologia de orientação a objetos. Desta forma, em termos de visão de implementação, os componentes Cosmos são mapeados em classes.

O ambiente Eclipse foi utilizado para o desenvolvimento do protótipo e da aplicação. A arquitetura utiliza a tecnologia Java RMI para dar suporte ao protocolo de comunicação entre configuradores, o protocolo UDP para transmissão de dados de fluxo e o *framework* JMF da Sun, para captura e apresentação de vídeo. A escolha da linguagem Java foi motivada pelo fato dela ser usada nas definições das APIs que compõem os principais padrões para os sistemas atuais de TVDI. Desta forma, pode-se viabilizar no futuro a portabilidade de aplicações desenvolvidas nestes modelos, consumindo um menor esforço para o processo de adaptação.

Um aspecto que deve ser destacado, apesar de mencionado anteriormente, é que este trabalho está direcionado para as questões de configuração e gerenciamento de recursos. Desta forma, o trabalho

não entra em detalhes a respeito de modelos e ferramentas para projeto, desenvolvimento e implementação de componentes do *framework* Cosmos e do *middleware* AdapTV. A arquitetura definida para desenvolvimento e configuração de aplicações pressupõe que os componentes podem ser desenvolvidos por terceiros de forma totalmente independente, apenas observando o modelo de componentes Cosmos.

5.2.3.1 Modelo de Implementação do *Parser*

As experiências adquiridas com a implementação inicial do *middleware* AdapTV [77] indicaram a necessidade de aumentar o nível de flexibilidade do modelo, tornando as definições do *parser* e da linguagem independentes da implementação do *middleware*. Na implementação mencionada, o *parser* foi definido no escopo do *Configurator*; assim, uma possível incorporação de um novo *parser* envolveria uma alteração no código do componente *Configurator*. Neste sentido, a arquitetura atual evoluiu para o modelo de classes apresentado na Fig. 5.5. Esta arquitetura considera o padrão de projeto *Abastract Factory* [127], servindo de base para a construção de *parsers* para ambientes baseados no Cosmos,

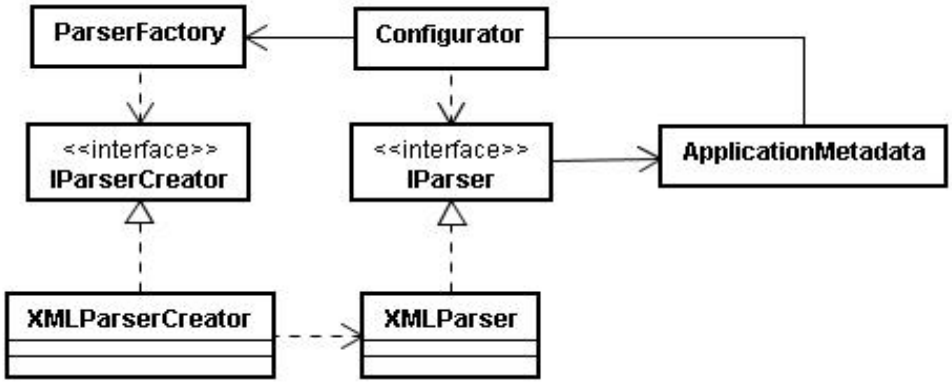


Fig. 5.5: Modelo para *Parser*

Um componente *parser* (no *middleware* AdapTV, o *parser XMLParser*) - componente que provê a interface *IParser* - , é o elemento responsável por gerar os metacomponentes associados à descrição da aplicação. Normalmente, ele realiza a sua tarefa processando um arquivo contendo uma especificação de configuração dos componentes de uma aplicação, devendo verificar a sintaxe da

especificação. Entretanto, o *parser* pode ser projetado para processar uma especificação gerada interativamente, possivelmente usando ferramentas de programação visual.

O resultado gerado por um *parser* em uma plataforma Cosmos é utilizado para construir uma estrutura composta de metadados denominada *ApplicationMetadata*, contendo os componentes, conexões e as respectivas configurações relacionadas na especificação.

Para dar início a um processo de configuração, cabe ao *Configurator* utilizar uma interface que dê acesso às fábricas de componentes *parsers* (interface *IParserCreator*) abstraindo-se, desta forma, de qual classe concreta o *parser* será instanciado. A Fig. 5.5 descreve este modelo seguindo o padrão de projeto *Abastract Factory* [127] que fornece uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

No modelo da Fig. 5.5 o *Configurator* utiliza a classe *ParserFactory* para criar uma fábrica específica (*XMLParserCreator*) que deve prover uma implementação para a interface *IParserCreator*. Por sua vez, classes que implementam a interface *IParserCreator* devem criar componentes do tipo *IParser*, os quais serão responsáveis pelo processamento dos arquivos de configuração. O *Configurator* tem acesso ao elemento *ApplicationMetadata* através da interface *IParser* do correspondente *Parser*. Como *ApplicationMetadata* contém a representação da configuração da aplicação, o *Configurator* pode realizar a configuração efetiva da aplicação e do *middleware* acessando os metadados correspondentes através da interface *IParser*.

A classe *ParserFactory* é responsável pela criação das fábricas de *parsers*; para isto, na descrição da classe correspondente da Fig. 5.6 define-se o método estático *parserCreate*, que recebe como parâmetro o tipo de fábrica a ser criada, retornando a correspondente fábrica para o configurador.

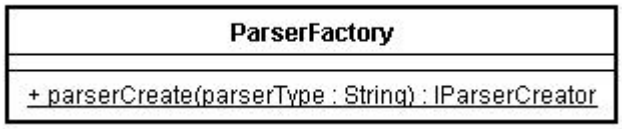


Fig. 5.6: Modelo da classe *ParserFactory*

A interface *IParserCreator* provê a funcionalidade de uma fábrica abstrata de fábricas de *parsers*. Sua descrição é mostrada na Fig. 5.7 indicando que a assinatura da operação *parserCreate* deve estar presente em todas implementações de fábricas de *parsers*. Sua função consiste em instanciar o *parser* especificado no parâmetro *parserType* da Fig. 5.6.

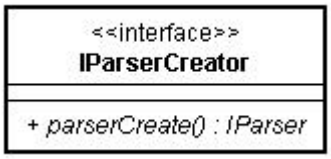


Fig. 5.7: Método da interface *IParserCreator*

Assim, para os *parsers* específicos poderem ser instanciados, eles devem implementar a interface *IParser* que é utilizada pelo *Configurador* para obter os metadados da aplicação. Essa interface tem o método *getApplication* que faz a leitura do arquivo de configuração e gera a hierarquia de metadados disponibilizando-a para o *Configurator*. A Fig. 5.8 mostra a especificação dessa interface.

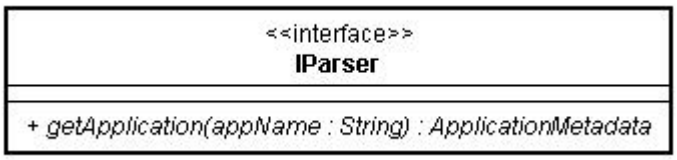


Fig. 5.8: Método da interface *IParser*

5.2.3.2 Modelo de Implementação do AdapTV

Esta seção apresenta os principais componentes da arquitetura de implementação definida para o protótipo do *middleware* AdapTV. No protótipo foram incorporados os componentes dos seguintes modelos do Cosmos:

- **gerenciamento de propriedades** - modelo de metacomponentes descrito na Seção 4.1.4;
- **gerenciamento de QoS** - modelo de QoS, descrito na Seção 4.2.3;
- **gerenciamento de interconexão** - modelo de interconexão, descrito na Seção 4.2.4.
- **gerenciamento de recursos** – modelo de configuração e gerenciamento de recursos descritos na Seção 4.3.

Os parágrafos seguintes abordam de forma abreviada as arquiteturas de implementação definidas para os modelos de QoS e de interconexão e, de forma mais detalhada, os modelos de metacomponentes e metaprogramação associados ao processo de configuração. O destaque para o modelo de matacomponentes deve-se à relevância do mesmo para o suporte à configuração e à adaptação dinâmica.

Processamento de metadados

Os elementos relacionados aos modelos de metadados, metacomponentes e *parser*, conforme ilustrado no diagrama de classes da Fig. 5.9, dão a base para a implementação do modelo de configuração definido pelo Cosmos, e por conseqüência, para o *middleware* AdapTV.

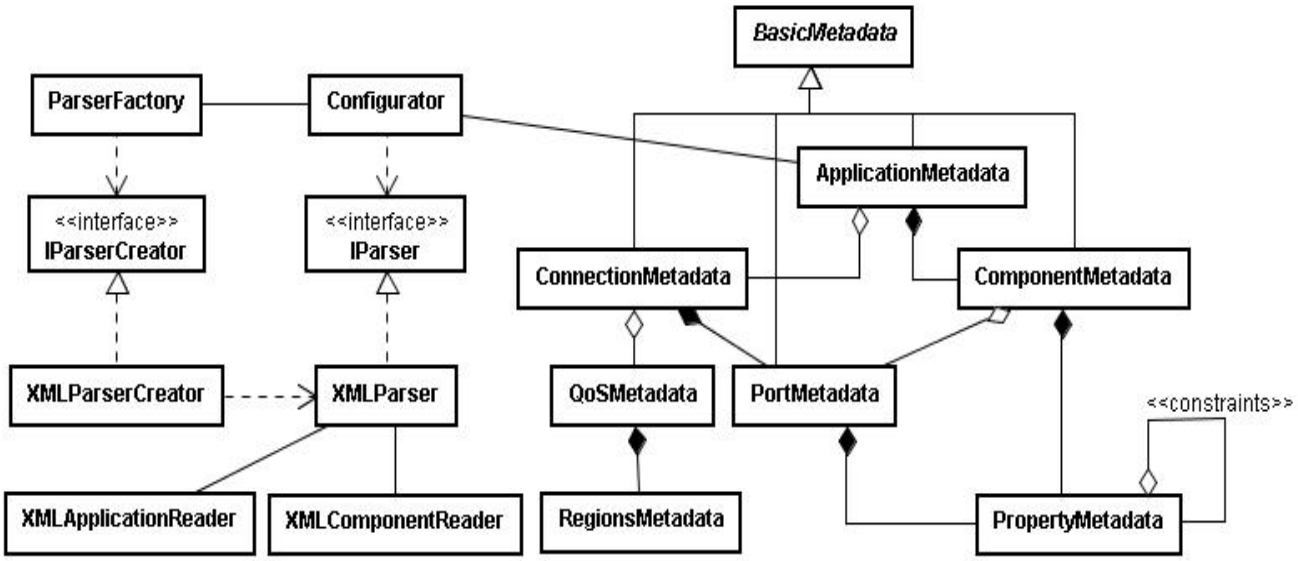


Fig. 5.9: Arquitetura de implementação do *framework* de configuração

A arquitetura de implementação introduzida na Fig. 5.9 considera os componentes do modelo de metacomponentes definidos na Fig. 4.9. Desta forma, para representar os metadados foram utilizados os conceitos de propriedades e restrições. Propriedades e restrições são manipuladas pelo *parser* e representadas no modelo de implementação por uma mesma classe, chamada no modelo como *PropertyMetadata*. Para os demais componentes, há uma classe para cada tipo de metadado, conforme pode ser observado no diagrama de classes da Fig. 5.9.

Para o *parser*, foi definida uma linguagem de especificação de aplicações baseada em XML de acordo com os modelos apresentados nas Seções 5.2.2.1 e 5.2.3.1. A implementação desta linguagem incorpora parte do código utilizado no protótipo inicial [77]. A Fig. 5.9 mostra o diagrama de classes associado à implementação atual, onde a hierarquia de composição ocorre apenas no nível da aplicação.

Para implementação de um *parser* específico de uma linguagem, deve-se implementar uma fábrica que implemente a interface *IParserCreator* e o respectivo *parser*, o qual deverá implementar a interface *IParser*; esses elementos correspondem, respectivamente, às classes *XMLParserCreator* e *XMLParser* da Fig. 5.9.

A classe *Configurator* é a classe responsável por iniciar, configurar e gerenciar os recursos e componentes da aplicação; ela é uma adaptação da classe utilizada na proposta inicial [77], de modo a introduzir os conceitos de metadados e *parser*.

Implementação do modelo de QoS

A implementação do modelo de QoS utiliza intensamente a hierarquia de classes definida na seção anterior para processamento de metadados. Um diagrama UML com uma visão simplificada envolvendo as classes de implementação associadas aos componentes do modelo de QoS [83] [84] é apresentado na Fig. 5.10.

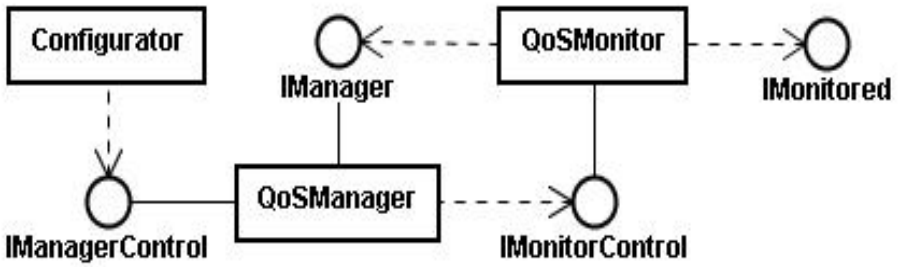


Fig. 5.10: Diagrama de classes simplificado para implementação do modelo de QoS

A classe QoS *Manager* implementa as interfaces *IManagerControl* e *IManager* descritas na seção 4.2.3. A interface *IMonitorControl* é implementada pela classe *QoSMonitor*; as operações desta interface encontram-se descritas na página 87. A interface *IStatusControl* foi definida no escopo do modelo de componentes Cosmos, ou seja, as operações desta interface devem ser implementadas em todos os componentes de um sistema baseado no *framework* Cosmos. A sua descrição é apresentada na página 64. A interface *IMonitored* deve ser implementada apenas em componentes que são passíveis de realização de monitoramento de seu estado. A descrição das operações associadas a esta interface encontra-se descrita na página 86.

Implementação do modelo de interconexão

Esta seção apresenta, de forma resumida, as principais classes que compõem a arquitetura de implementação do modelo de interconexão discutido na Seção 4.2.4. As principais classes definidas para implementação são apresentadas na Fig. 5.11.

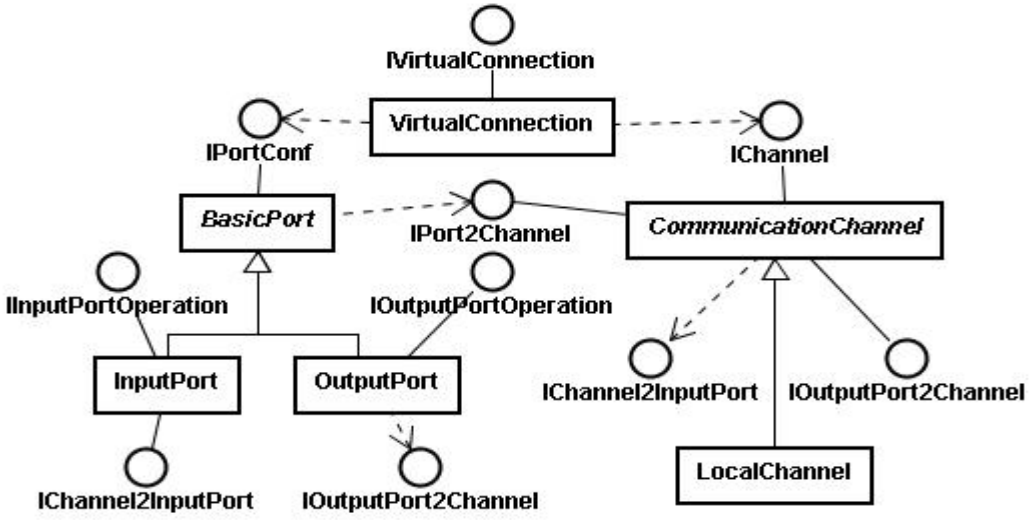


Fig. 5.11: Diagrama de classe com as principais classes do modelo de interconexão

A interface *IVirtualConnection*, utilizada pelo *Configurator* para gerenciar uma interconexão, é implementada pela classe *VirtualConnection*. Esta classe utiliza a hierarquia de metacomponentes (gerenciada pelo *ApplicationProxy*) para armazenar e manipular a lista de componentes associados (portas e canais), descrições dos correspondentes metadados e respectivas referências.

A interface *IPortConf* é utiliza pela conexão virtual para controlar as portas de comunicação, configurando seus canais de comunicação, ou iniciando uma adaptação. A classe abstrata *BasicPort* funciona como uma abstração para representar as portas, podendo ser especializada de acordo com o tipo da porta (entrada/saída); ela implementa a interface *IPortConf*, definindo as operações de configuração comuns.

A classe abstrata *CommunicationChannel* tem o objetivo de garantir a incorporação, por parte de um canal de comunicação, dos métodos de gerenciamento definidos na interface *IChannel*. É através desta interface que uma conexão virtual consegue acesso ao canal de comunicação. A classe *CommunicationChannel* também é uma classe abstrata, possibilitando assim representar diferentes tecnologias de comunicação para os canais de comunicação de uma maneira transparente.

As interfaces *IInputPortOperation* e *IOutputPortOperation*, assim como *IChannel2InputPort*, *IOutputPort2Channel* e *IPort2Channel*, foram descritas na Seção 4.2.4, sendo apresentadas na Fig. 5.11 apenas com o objetivo de colocar no diagrama de classes informações que refletem a implementação realizada.

O Apêndice C apresenta e discute brevemente cada uma das operações definidas nas classes introduzidas nas Fig. 5.9, 5.10 e 5.11.

5.2.4 Processo de Configuração no *Middleware* AdapTV

Esta seção apresenta as ações típicas de um processo de configuração do *middleware* e de aplicações no AdapTV. As ações são basicamente as seguintes: escolha e criação do *parser* a ser utilizado; processamento da especificação da aplicação (*parsing*); negociação e configuração de propriedades, instanciação dos componentes e configuração dos mesmos. O processo de configuração é introduzido através da descrição de alguns diagramas de seqüência UML envolvendo as principais interações entre os componentes da arquitetura.

A Fig. 5.12 ilustra o processo de criação do *parser* e o momento da aquisição, por parte do *Configurator*, do *ApplicationMetadata* que contém a especificação da aplicação, conforme definido pelo modelo de processamento dos metadados apresentado na Seção 5.2.3.2 e pela arquitetura de implementação do *parser* descrita na seção 5.2.3.1. Inicialmente, o *Configurator* solicita ao elemento *ParserFactory* para criar uma fábrica capaz de processar uma das linguagens definidas para o *middleware*. Seguindo o padrão *Abstract Factory*, o argumento da solicitação de criação é quem define o tipo de fábrica a ser criada. Esta fábrica implementa a interface *IParserCreator* utilizada pelo *Configurator* para acessar a fábrica recém instanciada. O *Configurator* solicita então a criação do *parser* correspondente. O acesso ao *parser* se dá através da interface *IParser*, usada pelo *Configurator* para recuperar os metadados que descrevem a aplicação. Estes metadados são alimentados pelo *parser* à medida que ele processa os arquivos contendo a especificação da aplicação.

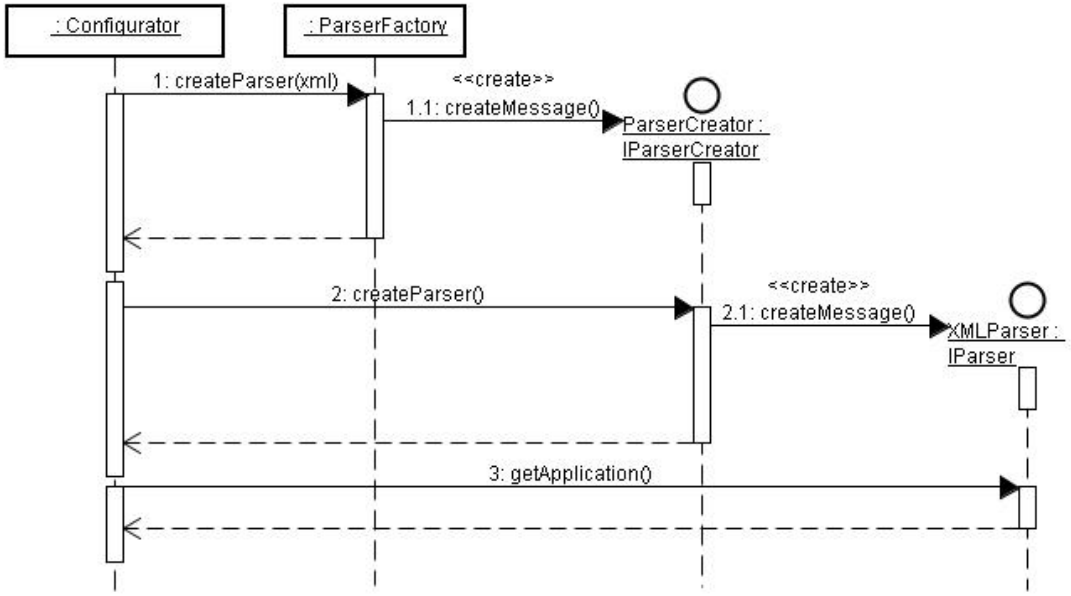


Fig. 5.12: Processo de criação do *parser*

A Fig. 5.13 apresenta uma visão do processo de negociação realizado na configuração dos componentes. Para desempenhar o seu papel, o *parser* XML utiliza dois componentes: o *XMLApplicationReader* e *XMLComponentReader* usados para fazer leituras de arquivos XML que contêm especificações das aplicações e dos componentes que as compõem (explicitados nas relações de dependências), respectivamente. Na Fig. 5.13, o *Configurator* solicita as conexões ao *parser XML* (método *getConnection* do componente *XMLApplicationReader*), define valores de parâmetros e propriedades usando operações para verificar o casamento de valores de propriedades comuns, através do método *matchProperties* [31], e cria as portas para os componentes envolvidos (método *createPorts* do componente *PortFactory*). Em seguida, o *Configurator* cria uma instância do componente *VirtualConnection* utilizada para representar a conexão. As portas previamente criadas são adicionadas ao componente *VirtualConnection* (método *attachPorts*), para então ser chamada a operação responsável por efetivar a conexão (método *connect*).

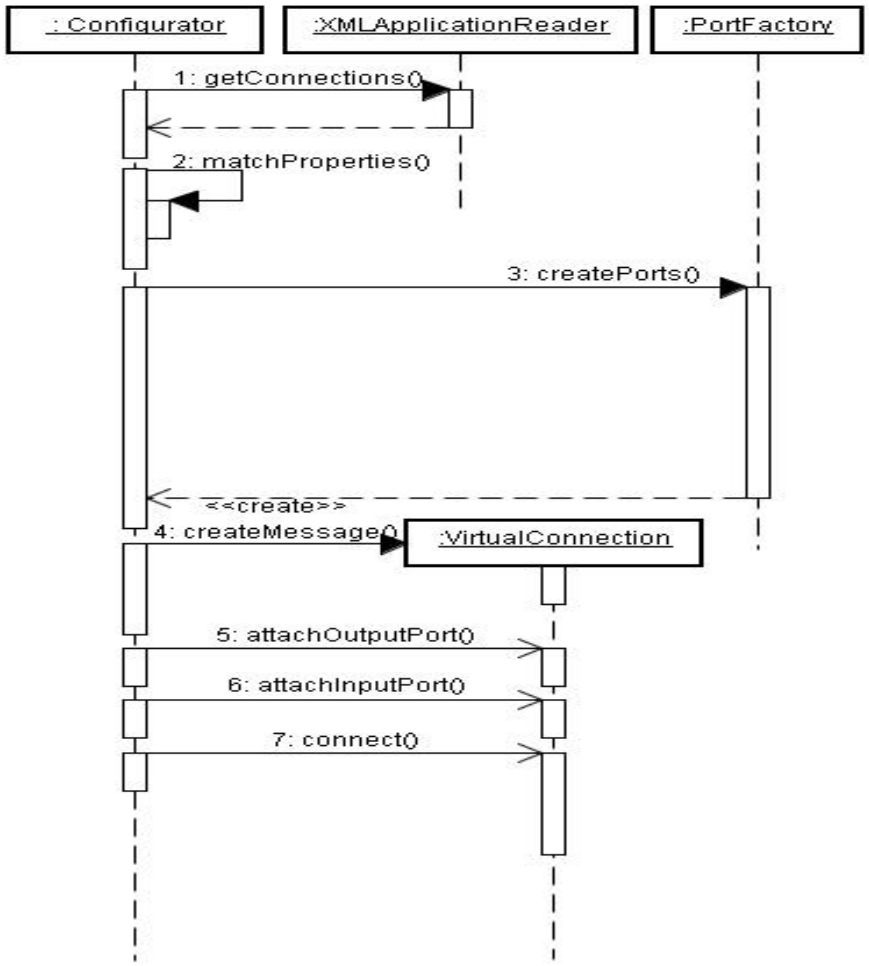


Fig. 5.13: Negociação e Ajuste de Propriedades em Interconexão de Componentes

Após ocorrer o processo de negociação e configuração de propriedades, é necessário obter os recursos necessários para o estabelecimento da interconexão. Através do componente *VirtualConnection*, o *Configurator* inicia o processo de instanciação de um canal utilizando como base o resultado da negociação de propriedades descrito anteriormente. A Fig. 5.14 apresenta uma visão geral do processo de interconexão sob o ponto de vista da conexão virtual. De posse dos metadados que descrevem as portas e a localização dos componentes envolvidos, a conexão virtual inicia o processo de interconexão. Após a instanciação das portas de comunicação em seus devidos domínios de endereçamento (lembrar que os metadados dos componentes de uma aplicação incluídas no *ApplicationProxy* residem no mesmo local que o *Configurator* mestre), a conexão virtual inicia a alocação do canal de comunicação adequado levando em consideração a localização dos componentes envolvidos. As condições utilizadas para a escolha da tecnologia de comunicação podem mudar de acordo com o ambiente de implementação do protótipo ou, dinamicamente, dependendo do estado da plataforma. Na implementação foi considerada a localização do componente com base no seu endereço IP. Para cada par de portas envolvidas, a conexão virtual instancia um canal de comunicação diferente, podendo inclusive envolver diferentes tecnologias simultaneamente.

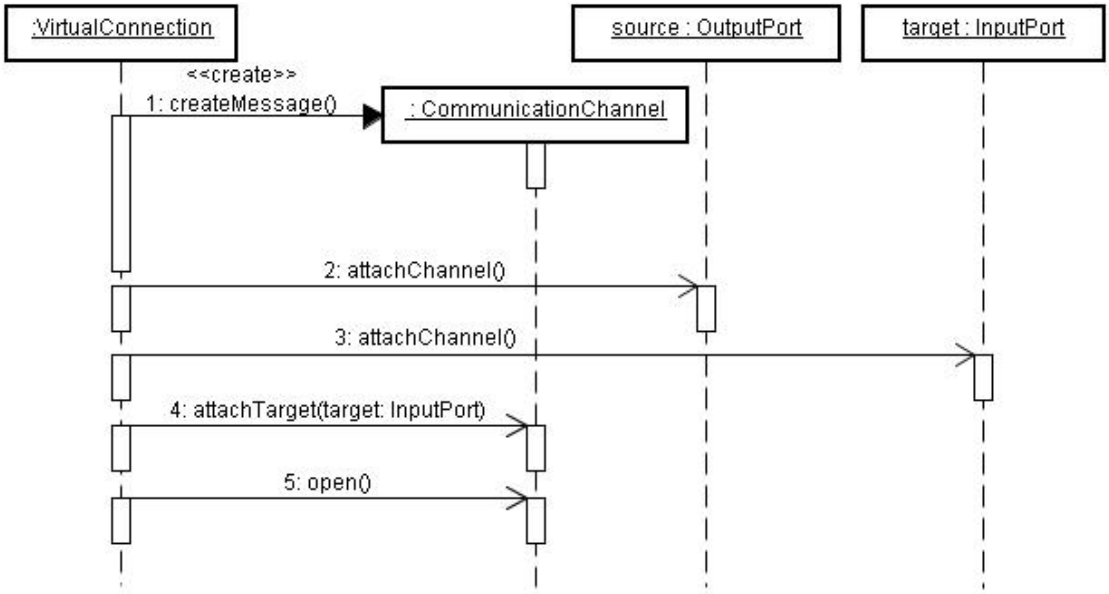


Fig. 5.14: Criação do canal no processo de interconexão.

Uma vez que o canal tenha sido instanciado, as portas de comunicação são configuradas de maneira a estabelecer uma vinculação com seu respectivo canal (chamadas *attachChannel*). Por sua

vez, o canal vai estabelecer um vínculo com uma referência para a porta de entrada (método *attachTarget*) na qual ele deverá colocar os dados associados.

A Fig. 5.15 ilustra um processo de adaptação reativa destacando as principais operações envolvidas. Esta figura apresenta uma versão simplificada da descrição detalhada na Fig. 4.36.

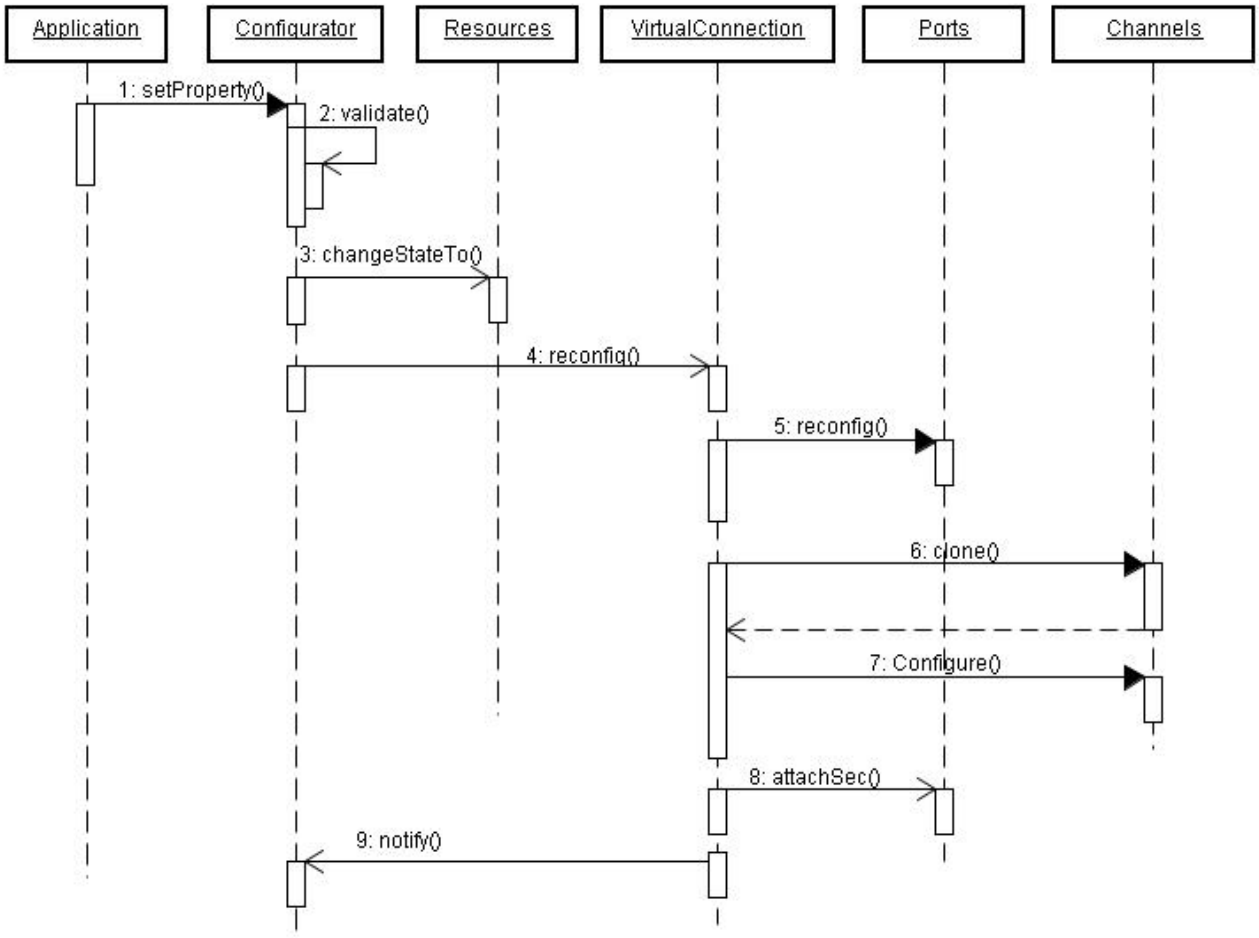


Fig. 5.15: Processo Simplificado de Adaptação Reativa

Na Fig. 5.15, a aplicação solicita uma adaptação ao *Configurator*, pedindo, por exemplo, para mudar alguma propriedade do sistema. O *Configurator* identifica os componentes afetados por esta mudança e inicia um novo processo de negociação para verificar se todos os componentes suportam a alteração da propriedade correspondente. Após esta verificação (operação *validate*), é iniciado o processo de adaptação. Nesse processo, o *Configurator* notifica os recursos envolvidos (operação *changeStateTo*) e a conexão virtual (operação *reconfig*) para que iniciem a reserva de recursos e se preparem para a reconfiguração. Estas ações ocorrem em paralelo com o funcionamento normal dos componentes envolvidos. A conexão virtual realiza a mesma operação nas portas envolvidas e dispara

uma operação para realizar uma clonagem do canal de comunicação em operação, de modo que o fluxo atual não seja interrompido durante a adaptação. As portas passam a possuir temporariamente dois canais cada uma, mas continuam a operar através do canal primário. Em seguida, o *Configurator* notifica os componentes que a adaptação pode ser efetivada. Neste momento, o(s) componente(s) transmissor(es), notifica(m) sua(s) porta(s) de saída e troca(m) seus parâmetros de operação. A porta de saída realiza a troca do canal de comunicação descartando o canal primário. Este último se encarrega de entregar o restante do fluxo que ainda se encontra em trânsito. A porta de entrada, por sua vez, começa a receber dados de dois canais de comunicação diferentes, salvando-os em *buffers* diferentes. Quando o *buffer* do canal primário se esvaziar por um pedido do componente receptor, a porta realiza a troca de canais, descartando o canal primário, que agora pode ser desalocado, e notifica o componente receptor associado para que ele altere seus parâmetros operacionais.

5.3 Aplicação-Exemplo Implementada

Com o objetivo de ilustrar o uso do protótipo, bem como demonstrar a viabilidade de implementação do Cosmos, esta seção descreve uma aplicação definida e implementada no contexto do *middleware* AdapTV numa plataforma distribuída.

5.3.1 Descrição da Aplicação

A aplicação definida envolve dois componentes recursos virtuais básicos distribuídos: o primeiro, responsável por capturar de um arquivo os dados de um fluxo de vídeo e enviá-los para a sua respectiva porta de saída, e o segundo, responsável por receber e exibir este fluxo. O componente produtor de fluxo, denominado *VideoFlowProducer*, pode gerar o fluxo no formato MPEG-1 ou MPEG-2 e transmitir com uma taxa que varia de 1 a 30 quadros por segundo. O processo de escolha do formato e do parâmetro de QoS (taxa de transmissão em quadros por segundo) envolve atividades de negociação de propriedades com a finalidade de obter valores compatíveis entre os requisitos especificados na aplicação, as capacidades dos componentes e o estado da plataforma. Uma simplificação adotada no protótipo para representar a variabilidade da QoS no CODEC, para efeito de codificação dos quadros do fluxo, foi considerar três arquivos de um mesmo vídeo codificados respectivamente com níveis de QoS considerados como *high*, *normal* e *low*.

Concomitantemente com a execução da aplicação - processos de captura, transmissão, recepção e apresentação do fluxo na plataforma distribuída -, o sistema realiza o monitoramento da QoS e, eventualmente, dispara adaptações com base na especificação de requisitos de QoS. Um processo de adaptação no exemplo envolve a troca do fluxo, ou seja, a mudança do arquivo utilizado no processo de captura (QoS diferente). O impacto desta mudança pode ser a perda ou melhoria da QoS do vídeo, dependendo da adaptação correspondente estar associada a um ajuste decorrente de sobrecarga da rede, ou de uma tentativa de retomada da QoS escolhida pela aplicação, após o pico da sobrecarga. Como no protótipo o processo de codificação foi abstraído através do uso de arquivos codificados previamente, o monitoramento da taxa de transmissão fica limitado devido à simplificação. Assim, com o objetivo de tratar o processo como uma aproximação de uma situação onde existe a possibilidade de considerar a flutuação da QoS no CODEC, adotou-se que o recurso virtual associado ao CODEC, quando monitorado, estabelece randomicamente uma taxa de geração e transmissão atual, de forma a provocar necessidades de adaptações dinâmicas.

O exemplo, apesar de simples, explora os principais aspectos definidos no Cosmos, como o desenvolvimento de recursos virtuais, especificação de aplicações, *parsing*, instanciação de componentes, negociação de propriedades, execução em ambientes distribuídos, monitoramento de QoS e adaptação dinâmica através da reconfiguração reativa e pró-ativa.

5.3.2 Arquitetura da Aplicação

A Fig. 5.16. apresenta um diagrama de componentes com uma visão geral da aplicação. No diagrama, são mostrados os dois componentes recursos virtuais, *VideoFlowProducer* e *VideoFlowConsumer* e suas respectivas portas, *OutputFlow* e *InputFlow*, que são exploradas pelos componentes para enviar e receber dados de fluxo. Destacam-se também na Fig. 5.16 as especificações XML da aplicação e dos recursos virtuais *VideoFlowProducer* e *VideoFlowConsumer,* bem como a descrição dos metadados correspondentes à especificação da aplicação, representada e gerenciada no *middleware* pelo componente *ApplicationProxy*.

O componente *ApplicationProxy* define uma estrutura independente de tecnologia capaz de representar os metadados dos componentes da aplicação. Assim, a criação de uma aplicação envolve o processamento da especificação baseada em XML correspondente, onde o papel do *parser* consiste em gerar os respectivos metadados a serem introduzidos no *ApplicationProxy*.

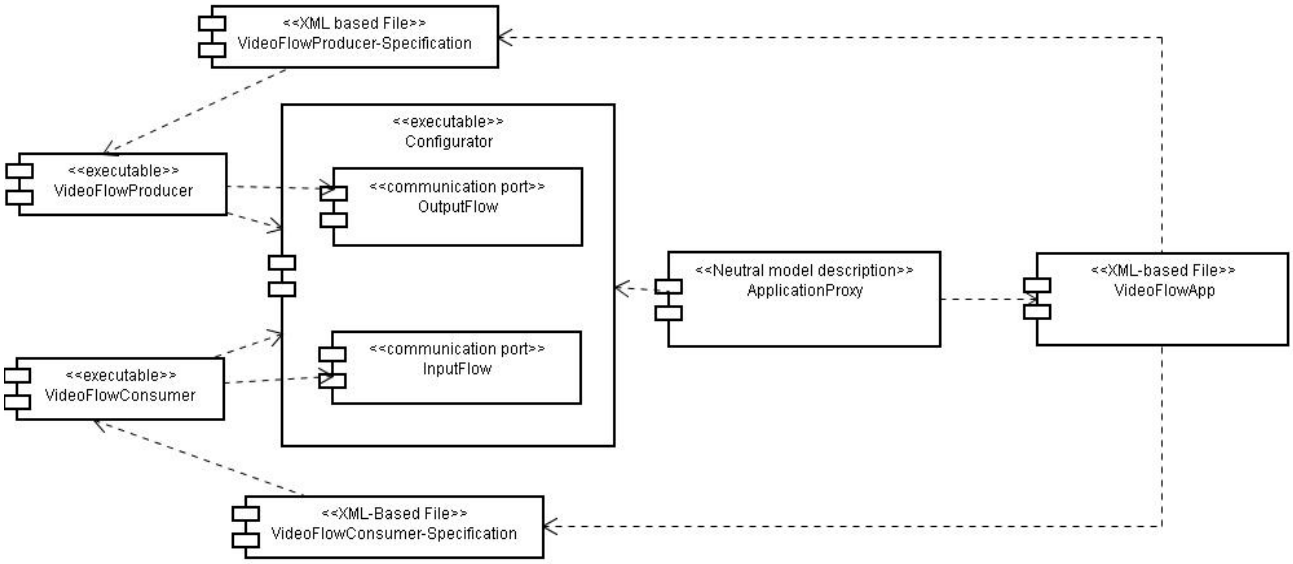


Fig. 5.16: Diagrama de componentes da aplicação-exemplo

A especificação XML é processada de acordo com o modelo descrito na Seção 5.2.3.1. A partir das informações geradas pelo *parser*, o *Configurator* realiza a instanciação dos componentes associados, bem como a configuração dos mesmos, onde restrições sobre o conjunto de possíveis valores de propriedades podem ser aplicadas, conforme a discussão sobre as ações envolvidas no processo de configuração no *middleware* AdapTV (Seção 5.2.4). Para isso, cada componente disponibiliza sua especificação XML, que é tratada pelo *Configurator* para realizar operações tais como: busca de componentes na hierarquia de dependência estabelecida; instanciação dos mesmos através das respectivas *Factories*; ajustes de propriedades para garantir os requisitos especificados e realização de conexões entre portas. Uma visão geral sobre o processo de negociação e ajuste de propriedades de componentes foi apresentada na Fig. 4.35, onde é ilustrado o processo de interconexão de componentes.

É importante frisar que as portas são alocadas e mantidas pelos *Configurators* em seus respectivos espaços de endereçamento. Desta forma, os componentes remotos têm suas portas alocadas nos *Configurators* escravos correspondentes. A alocação ocorre durante o processamento da especificação, após o processo de negociação. Para os componentes produtor e consumidor de fluxo operarem, eles precisam ter durante a fase de execução as referência de suas portas. Para ilustrar como ocorre este processo, as principais interações envolvidas são mostradas no diagrama de seqüência da Fig. 5.17.

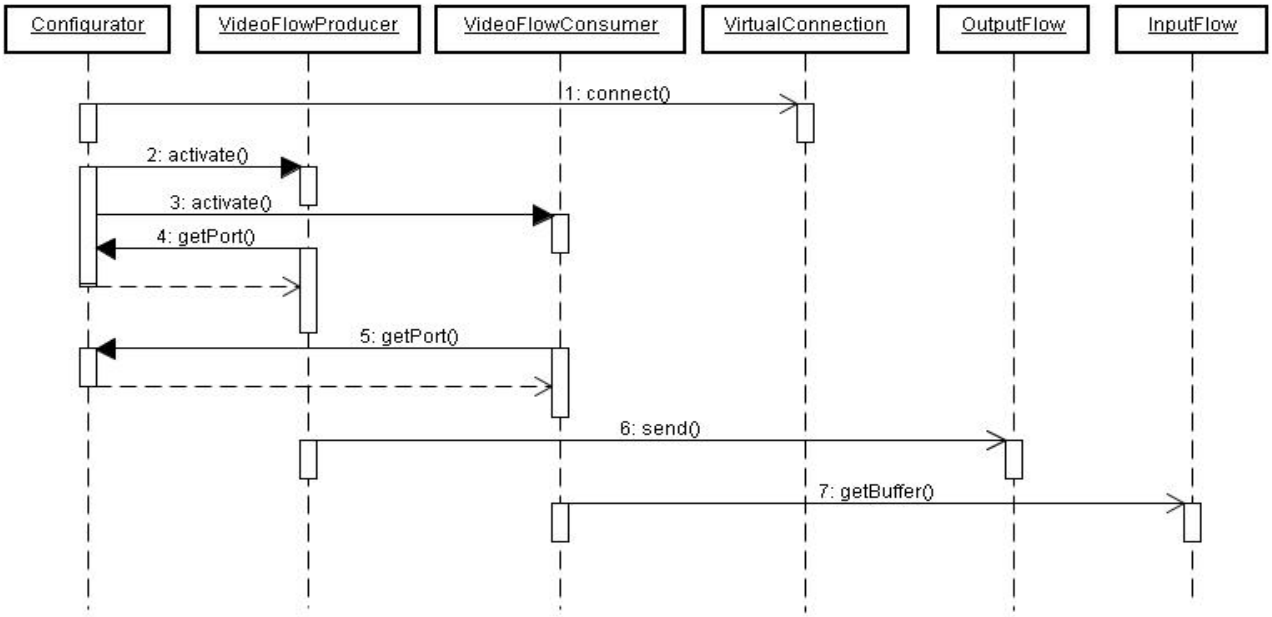


Fig. 5.17**:** Comunicação entre portas

Na figura são abstraídas as ações anteriores relacionadas com negociação e criação do canal de comunicação da operação *connect*; estas ações foram ilustradas, respectivamente, na Fig. 5.13 e Fig. 5.14. A Fig. 5.17 supõe que os recursos requeridos foram obtidos, de forma que a análise começa a partir da ativação dos componentes *VideoFlowProducer* e *VideoFlowConsumer* através das chamadas da operação *activate*. Esta operação é disparada pelo *Configurator* através da indicação ao *ApplicationProxy* (não explicitado na figura), que se encarrega de repassar a atualização para os respectivos componentes.

Para um componente utilizar uma porta, ele precisa requisitar ao *Configurator* do seu domínio uma referência à porta através do método *getPort()*; o *Configurator* verifica se o solicitante é o mesmo componente que especificou a porta através do XML, retornando-a em caso afirmativo. De posse da referência, considerando o exemplo explorado na Fig. 5.17, as únicas interfaces da porta que interessam para os componentes recursos virtuais *VideoFlowProducer* e *VideoFlowConsumer* são, respectivamente, as interfaces operacionais *IOutputPortOperation* e *IInputPortOperation*, cujas descrições são apresentadas na Fig. 4.30.

A Fig. 5.18 mostra a especificação XML do componente *VideoFlowProducer*. Na especificação são explicitadas as capacidades e propriedades do fluxo produzido pelo componente. O componente *VideoFlowConsumer* não é apresentado porque possui uma descrição semelhante.

```
<COMPONENT
        name="br.natalnet.adaptv.VideoFlowProducer"
         description="Componente responsável por enviar dados de vídeo">
        <ATTRIBUTES>
                <ATTRIBUTE name="Title" default="Tramissor de Vídeo"/>
        </ATTRIBUTES>
        <PROPERTIES>
                <PROPERTY name="AllocatedMemory" default="10000"/>
        </PROPERTIES>
        <PORTS>
                <PORT name="OutputFlow"  TYPE "outputport"/>
                        <PROPERTY name="framerate" values="1..30"
                                        order="asc"/>
                        <PROPERTY name="encoding" values="MPEG-1, MPEG-2"/>
                </PORT>
        </PORTS>
</COMPONENT>
```

Fig. 5.18: Descrição XML do Componente *VideoFlowProducer*

A Fig. 5.19 mostra a especificação de uma configuração exemplo para a aplicação *VideoApp*. Nesta aplicação, o formato MPEG-2 do componente *VideoFlowProducer* foi restringido, conforme indicado na *tag* CONSTRAINT associada à porta *OutputFlow*. Assim, este formato não é considerado pelo *Configurator* no processo de negociação de propriedades. O exemplo também trata questões de gerenciamento da QoS ao associar o parâmetro de QoS *QoSBandwidth* à conexão virtual; conforme o modelo definido no Cosmos, a conexão virtual é o componente responsável por gerenciar os elementos que dão suporte à comunicação, os quais estão sujeitos a constantes variações de carga e atrasos. Na aplicação, o modelo de suporte à adaptação proposto é usado para a realização de adaptações envolvendo mudança de propriedades do fluxo (*framerate*) durante a operação do sistema.

Conforme a descrição da Fig. 5.19, o exemplo consiste de um componente produtor, cujo papel é distribuir vídeos, e um componente consumidor para receber e exibir os mesmos. O produtor tem capacidade para oferecer vídeos com diferentes níveis de QoS, onde cada vídeo pode ser transmitido em várias taxas (*frames* por segundo). Com o objetivo de facilitar o gerenciamento, a aplicação definiu três regiões, denominadas de acordo com os níveis das taxas de QoS (*framerate*) como *low*, *normal* e *high*. O cliente também pode exibir os fluxos de vídeo com diferentes taxas. A escolha da região ocorre no nível da configuração de alguns parâmetros internos. Cada região é caracterizada por uma faixa de valores (*range*) e um valor *default* para a propriedade *framerate*.

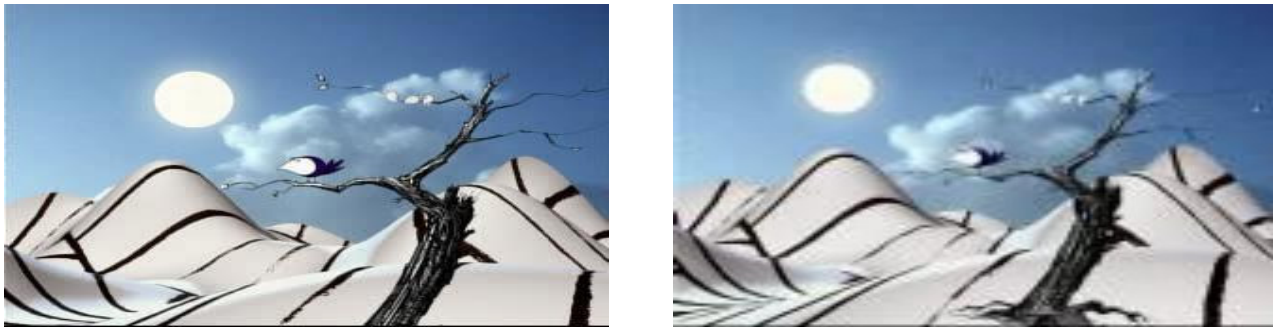
Como descrito na Seção 4.2.3 e ilustrado na Fig. 4.27, quando a especificação define requisitos de QoS, a arquitetura utiliza, durante a execução, componentes monitores para observar se os parâmetros especificados se mantêm dentro das faixas estabelecidas. Baseados neste modelo, os parágrafos seguintes dão uma breve explicação sobre a adaptação explorada no presente exemplo.

```
<APPLICATI0N name="VideoFlowApp">
<DEPENDS>
  <COMPONENT name="VideoFlowProducer"
             instance="videoFlowProducer"
             location=comp1,services.natalnet.br:8080/enquiring,
                      services.dimap.ufrn.br/enquiring">
    <PROPERTIES>
      <PROPERTY name="AllocatedMemory" value="5000"/>
    </PROPERTIES>
    <ATTRIBUTES>
      <ATTRIBUTE name="Title" value="Example1"/>
    </ATTRIBUTES>
    <PORTS>
      <PORT name="OutputFlow">
        <CONSTRAINT property="Encoding" remove="MPEG-2"/>
      </PORT>
    </PORTS>
  </COMPONENT>
  <COMPONENT name="VideoFlowConsumer"
             instance="videoFlowConsumer"
             location= comp2,services.natalnet.br/enquiring,
             services.dimap.ufrn.br/enquiring">
    <PROPERTIES>
      <PROPERTY name="AllocatedMemory" value="5000"/>
    </PROPERTIES>
    <ATTRIBUTES>
      <ATTRIBUTE name="Title" value="Example2 "/>
      </ATTRIBUTES>
  </COMPONENT>
</DEPENDS>
<CONNECTIONS>
  <CONNECT from="br.natalnet.adaptv.videoFlowProducer:OutputFlow"
           to="br.natalnet.adaptv.videoFlowConsumer:InputFlow">
    <QOS parameter = "QoSBandwidth">
      <DEFAULTREGION> High </DEFAULTREGION >
      <FREQUENCY> 100.00</FREQUENCY>
      <TESTTIME> 1000.00</TESTTIME>
      <REGIONS>
        <"REGION name =  "High" >
          <RANGE min =  "30"/>
          <PROPERTY_DEFAULT name="framerate" values = "30"/>
        </REGION>
        <REGION name =  "Normal" >
          <RANGE max = "29  min =  "10"/>
          <PROPERTY_DEFAULT name="framerate" values = "18"/>
        </REGION
        <REGION name =  "Low" >
          <RANGE max =  "9"/>
          <PROPERTY_DEFAULT name="framerate" values = "7"/>
        </REGION
      </REGIONS>
    </QOS>
  </CONNECT>
</CONNECTIONS>
</APPLICATI0N>
```

Fig. 5.19: Descrição XML da configuração da aplicaçãoVideoFlowApp

De acordo com a especificação, a aplicação começa a execução na região de QoS denominada ***High***, configurada na especificação como a região de operação *default* através da *tag* DEFAULTREGION. Nesta região, o valor de referência *default* para início de operação com relação à propriedade *framerate* configurada corresponde a 30 quadros por segundo. Se o monitor associado perceber alguma degradação na taxa de transmissão, ele aciona o correspondente gerente de QoS, que indica uma nova região (provavelmente a região chamada como ***Normal***), com a propriedade *framerate*

configurada para operar inicialmente com a taxa de 18 quadros por segundo, correspondendo ao valor especificado como *default* para a propriedade *framerate* nesta região. Conforme mencionado, no protótipo o monitor solicita as informações da taxa atualizada ao componente recurso virtual *FlowProducer*, que fornece um valor randômico. Feita a negociação, o *middleware* faz a troca das propriedades do fluxo; na verdade, na implementação realizada, o processo produtor de vídeo começa a capturar os quadros do fluxo do arquivo que utiliza a codificação de vídeo em um formato diferente, correspondendo na aplicação à propriedade de QoS denominada como *Normal*. A mudança de fluxos é realizada de forma praticamente imperceptível, a menos do efeito visual relacionado com a diminuição da qualidade, conforme pode ser observado na transição da QoS *High* para *Normal* mostrada na Fig. 5.20, onde são apresentadas duas telas extraídas do fluxo correspondente à execução da aplicação-exemplo no momento da transição da QoS .



QoSBandwidth = High **QoSBandwidth = Normal**

Fig. 5.20: Exemplos de telas com a seqüência de imagens na transição da QoS

De acordo com o parâmetro TESTTIME, o gerente de QoS deve tentar periodicamente recompor o nível da QoS para a região *default*, ou pelo menos, para alguma região de QoS melhor que a região atual, quando provavelmente o pico de sobrecarga da rede terá passado. Desta forma, o *middleware* realiza de forma transparente o processo de adaptação. Para isso, ele precisa informar ao componente produtor que ele deve modificar a qualidade do fluxo de vídeo. Por outro lado, o consumidor também precisa ser notificado da mudança, de modo a poder interpretar corretamente o fluxo recebido. As principais partes da especificação de uma conexão, neste exemplo, são apresentadas na Fig. 5.19. Neste exemplo, a adaptação ocorre no nível interno, mudando os valores das propriedades do próprio componente, sem necessitar trocar componentes.

No protótipo também foram realizadas reconfigurações reativas. Para controlar o momento da adaptação no processo de adaptação reativa, foi implementada no protótipo uma interface visual onde o

usuário seleciona a mudança de propriedades durante a operação da aplicação Deve-se lembrar que adaptações reativas somente poderão ser realizadas com a autorização (validação) do *Configurator* através de chamadas a operações da interface *IUsersSets* (Fig. 4.18). Desta forma, o *Configurator* pode verificar se é possível realizar a adaptação considerando o estado atual dos recursos da plataforma.

5.3.3 Aspectos da Implementação Distribuída

Os componentes foram implementados no protótipo como classes Java e, para comunicação entre configuradores remotos, foram utilizados os mecanismos de programação baseados em chamadas remotas de procedimentos (RMI); para transmissão de dados de fluxo de vídeo, os mecanismos empregados foram baseados em *soquetes*-UDP.

Uma vez que o *Configurator* é o responsável pela negociação e instanciação dos componentes, deve existir uma comunicação entre os *Configurators* envolvidos. Esta comunicação trata, por exemplo, aspectos relacionados com alocação de recursos, negociação de propriedades e instanciação dos componentes. Uma vez que o *Configurator* é quem controla seu espaço de endereçamento, somente ele pode verificar se uma determinada configuração é corente no contexto do ambiente distribuído.

Como o *ApplicationProxy* tem um papel similar ao de um repositório de metainformações, ele precisa ser acessado remotamente por componentes distribuídos da aplicação. Considerando então que há comunicações entre *Configurators*, precisava-se definir o comportamento do *ApplicationProxy* para tratar esta situação. Na implementação foi mantida a nomenclatura definida no Cosmos, em que o *Configurator* que realiza o processamento da especificação é denominado *Configurator* mestre; neste caso, o *ApplicationProxy* da aplicação fica no espaço de endereçamento deste *Configurator*. Durante o funcionamento distribuído, outros *Configurators* poderão estar envolvidos no ambiente; estes *Configurators* remotos são classificados como escravos. Os *Configurators* escravos precisam ter acesso ao *ApplicationProxy*, de forma que possam realizar consultas sobre as metainformações dos componentes correspondentes que estão em seus espaços de endereçamento, bem como de propriedades que possam alterar o funcionamento dos componentes sob sua responsabilidade.

Um componente remoto (*virtualResource*), isto é, aquele que é executado sob o espaço de endereçamento de um *Configurator* escravo, precisa estar registrado junto ao *ApplicationProxy* que fica sob o controle do *Configurator* mestre. Isso acontece porque o *ApplicationProxy* mantém uma referência para todos os componentes (recursos virtuais) da aplicação. A troca de uma propriedade em

um componente precisa passar pelo *ApplicationProxy* para que se possa validar esta mudança através de uma negociação de propriedades com os componentes envolvidos.

Da mesma maneira que um recurso remoto funciona sob o espaço de endereçamento de um *Configurator* escravo, as portas de comunicação associadas a este componente também o fazem. Uma conexão virtual, por sua vez, precisa ter acesso às referências das portas de comunicação envolvidas, fazendo-se necessário ter também acesso remoto para as portas de comunicação, de modo a estabelecer o controle sobre as mesmas.

Considerando os aspectos acima, percebe-se a necessidade da definição de uma espécie de protocolo de comunicação entre os *Configurators* envolvidos para que se possa alcançar o funcionamento distribuído de forma consistente. A comunicação entre *Configurators* se dá através da interface *IRemoteConfiguration* (definida na Fig. 4.15). Esta interface contempla todas as operações de comunicação entre os *Configurators* envolvendo, por exemplo, a criação de componentes, a validação de configurações através de metadados e a notificação de sucesso ou falha em determinada operação.

O acesso ao *ApplicationProxy* por parte do *Configurator* escravo se dá através da interface *IIntrospection* (descrita na Fig. 4.20), bastando para isso fazer com que o componente que implementa esta interface utilize as facilidades disponibilizadas pela tecnologia java RMI.

Para mudar o comportamento de um componente remoto, o *Configurator* mestre indica a mudança acessando e atualizando o componente *ApplicationProxy*; a mudança da propriedade associada é acionada usando uma referência de acesso à interface *IPropertyUpdate* (Fig. 4.7) do respectivo componente, observando o suporte aos requisitos de funcionamento distribuído.

O acesso a uma porta de comunicação (para fins de gerenciamento) só é realizado através de uma conexão virtual; para isso, ela utiliza a interface *IPortConf* (descrição na Fig. 4.32).

5.4 Considerações Finais

A implementação do protótipo do *middleware* e de um exemplo de aplicação mostram o potencial da arquitetura de configuração definida pelo *framework* proposto. Os testes realizados envolvendo adaptação demonstraram a viabilidade do modelo. Adicionalmente, eles contribuíram, através da experiência adquirida, para conceber uma visão integrada de todo o sistema, ficando assim estabelecidas claramente as etapas do processo de desenvolvimento.

Para o desenvolvimento do protótipo e da aplicação foi utilizada a ferramenta Eclipse. O desenvolvimento envolveu um esforço de programação de cinco colaboradores durante um ano, sendo quatro deles no nível de graduação e um no nível de mestrado. Atualmente, o protótipo conta com aproximadamente 100 classes, tendo cerca de 10000 linhas de código Java.

Capítulo 6

Conclusões e Considerações Finais

Vários trabalhos têm sido apresentados na literatura com o objetivo de solucionar problemas relacionados com desenvolvimento, disponibilização e execução de sistemas multimídia em plataformas distribuídas. Algumas das idéias exploradas por estes trabalhos foram citadas no decorrer do presente texto. Uma característica comum observada é a introdução de soluções na camada de *middleware* para gerenciamento de QoS e adaptação. Entretanto, o texto destaca que muitas das soluções ainda são limitadas. Neste contexto, a presente tese propõe o *framework* Cosmos.

Como atividade preliminar, o trabalho identificou os principais requisitos de suporte ao tipo de sistema em questão. Uma breve discussão sobre estes requisitos foi apresentada na Seção 2.3, onde foram abordados os seguintes aspectos: conectividade de componentes em plataformas distribuídas abertas; abordagens arquiteturais para gerenciamento da complexidade; suporte à adaptação e QoS e capacidades para realizar configuração e reconfiguração dinâmicas.

Para tratar estes requisitos, o Cosmos explora a tecnologia de desenvolvimento baseada em componentes de *software* e incorpora algumas características importantes de algumas arquiteturas apresentadas na literatura, identificadas e discutidas no Capítulo 3 no contexto do estado da arte [7] [8] [20] [29] [100].

A seguir, o capítulo faz uma retrospectiva resumida do trabalho destacando as principais contribuições e realizando uma análise qualitativa do *framework*; ao final, algumas idéias em termos de trabalhos futuros são discutidas.

6.1 Principais Contribuições

Como contribuição central do trabalho, destaca-se o *framework* Cosmos. O Cosmos define componentes abstratos tais como recursos virtuais, conexões virtuais e portas, os quais fornecem uma

visão geral da arquitetura de um sistema multimídia distribuído abstraindo detalhes de implementação e especificidades de tecnologia. De acordo com o modelo definido no Cosmos, a configuração de uma aplicação e de seus componentes ocorre através do processamento de uma especificação; nesta especificação são descritas as características dos componentes e os requisitos da aplicação. Uma vez processada a especificação, a descrição dessas características e requisitos é mantida em componentes do *middleware* (metacomponentes), podendo ser consultada e utilizada pelo *middleware*, ou pela própria aplicação, para realização de adaptações dinâmicas.

Um dos objetivos do Cosmos é atuar como um *framework* genérico, tipo "caixa cinza", que pode ser usado para o projeto e desenvolvimento da camada de *middleware* de uma variedade de sistemas multimídia distribuídos. Para atingir este objetivo, Cosmos define componentes abstratos correspondentes aos elementos arquiteturais do sistema. Estes componentes podem ser reutilizados e especializados. Adaptações evolucionais se dão internamente nos componentes através da configuração de propriedades. O processo de configuração e ajuste de propriedades leva em consideração a especificação com os requisitos da aplicação e as capacidades da plataforma onde os componentes da aplicação são instanciados.

O conceito de portas, amplamente explorado pela engenharia de *software* em vários contextos, foi introduzido no *framework* para suportar interações envolvendo fluxos contínuos entre componentes recursos virtuais. Um recurso pode definir várias portas, cada uma tendo um tipo associado. De modo a suportar diferentes tipos de interação, o *framework* definiu diferentes tipos de interface para portas, onde cada tipo provê uma semântica de operação distinta. Recursos locais ou remotos podem ser interconectados através de ligações entre portas, que são gerenciadas por componentes denominados *virtual connections* (conexão virtual).

O *framework* definiu um componente central, denominado *Configurator*, que tem a responsabildiade de iniciar, configurar e gerenciar os recursos e componentes do sistema (camadas de *middleware* e aplicação).

De acordo com o modelo definido pelo Cosmos, cada aplicação e componente precisa ser especificado utilizando uma linguagem de descrição arquitetural. Esta linguagem tem um papel similar à ADL. Assim, esta linguagem é usada para descrever a especificação de aplicações indicando os componentes requeridos, as conexões, os parâmetros e as propriedades associadas, assim como os requisitos de QoS necessários à sua execução. Baseado nesta especificação, o *Configurator* localiza e aciona fábricas capazes de instanciar os respectivos componentes. Na verdade, o *framework* não define

uma tecnologia, nem tampouco uma linguagem específica para descrição de aplicações e componentes. Ele apresenta um modelo de descrição independente de tecnologias, com regras e relacionamentos comuns para definição de aplicações e componentes. Este modelo apresenta uma estrutura similar à definida pela OMG para descrever a recomendação RAS [150].

Sob a visão de um projetista e desenvolvedor de componentes, é preciso seguir as prescrições (descrições das interfaces) dos componentes associados ao modelo básico de componentes Cosmos e conhecer os relacionamentos entre os elementos arquiteturais do *framework*. Para distribuir um componente desenvolvido, é preciso também disponibilizar uma metadescrição do mesmo – especificação com as características e propriedades do componente. A tarefa de usar este componente para realizar a composição da aplicação e, eventualmente, a interconexão dele com outros componentes do sistema, é de total responsabilidade do *middeware* através de seu componente central – o *Configurator*, observando a descrição da aplicação fornecida.

O modelo de metacomponentes definido no *framework* estabelece um mecanismo que permite tratar separadamente as propriedades dos componentes em relação à implementação dos mesmos. Esta abordagem provê flexibilidade para integração de componentes desenvolvidos em diferentes linguagens de programação e em diferentes tecnologias. Para isto, uma camada de portabilidade pode ser construída introduzindo código de adaptação durante o processo de configuração de acordo com as propriedades definidas nos metacomponentes. Este tipo de abordagem dá suporte à adaptação evolucional no Cosmos. Outra forma de adaptação tratada no Cosmos consiste em trocar dinamicamente os formatos de componentes de mídia através de reconfigurações baseadas nos metadados.. No Cosmos, metacomponentes são componentes associados aos componentes da arquitetura, onde são mantidas as características e especificidades descritas nas suas correspondentes especificações. O *Configurator*, em tempo de instanciação e de execução da aplicação, consegue descobrir as características de cada componente do sistema através de consultas a estes metacomponentes. Desta forma, o *Configurator* obtém os metadados que ele vai utilizar para realizar possíveis adaptações, de modo que os componentes possam ser executados na plataforma de acordo com os requisitos especificados na aplicação. Uma adaptação no *framework* é realizada explorando os conceitos de reflexividade. Operações da API definida para reflexividade dão suporte à realização de consultas e atualizações de metacomponentes. Os metacomponentes são mantidos num componente denominado *ApplicationProxy,* cujo papel consiste em preservar o conhecimento e a consistência da aplicação no contexto do *middleware*.

Além do *framework* Cosmos, pode-se destacar como contribuições desta tese, as definições da linguagem de especificação e dos modelos de matacomponentes, de QoS e de interconexão de componentes.

Por fim, outra contribuição a ser destacada é o *middleware* AdapTV. O desenvolvimento deste *middleware* foi realizado utilizando os componentes arquiteturais básicos definidos no *framework* Cosmos. O esforço associado ao desenvolvimento teve como objetivo mostrar a viabilidade de implementação do *framework* Cosmos. Adicionalmente, a arquitetura definida pelo AdapTV apresenta uma visão inovadora para plataformas de *middleware* de sistemas de TVDI; nesta visão, aspectos de adaptação dinâmica reativa e pró-ativa podem ser tratados usando componentes monitores de QoS para realizar a observação do comportamento do canal de comunicação.

6.2 Avaliação Qualitativa

O *framework* Cosmos incorpora muitos dos conceitos relacionados a plataformas de *middleware* adaptativos. Entretanto, considerando a comparação do Cosmos com algumas abordagens relacionadas na Seção 6.2.1, o Cosmos apresenta uma inovação em relação a suporte de configuração e gerenciamento de aplicações em sistemas multimídia distribuídos. Para prover suporte à reusabilidade, o Cosmos define componentes básicos com funcionalidades gerais comuns a plataformas de *middleware* para sistemas multimídia distribuídos e um modelo de suporte a adaptações evolucionais, permitindo a construção de novos componentes e serviços com uma abordagem arquitetural. Estes componentes podem ser configurados para domínios específicos ajustando valores de propriedades de acordo com os requisitos da aplicação e as características do *hardware* e *softwares* básicos da plataforma. O protótipo de uma plataforma de *middleware* para sistemas de TVDI, o *middleware* AdapTV, foi desenvolvido a partir dos componentes básicos do Cosmos. O reuso dos elementos do Cosmos na construção do *middleware* AdapTV está associado aos seguintes componentes:

- Componente abstrato *CosmosComponent*, envolvendo as implementações das operações para gerenciamento de propriedades e ciclo de vida;
- Componente *Configurator*, envolvendo as funcionalidades para gerenciamento de configuração;
- Componentes associados ao modelo de metacomponentes;
- Componentes associados ao modelo de QoS;
- Funcionalidades dos componentes do modelo de interconexão independentes de tecnologia.

Assim, analisando o processo de desenvolvimento e implementação do *middleware* AdapTV, pode-se considerar como bom o nível de reuso em relação aos componentes do *framework* Cosmos.

Em relação ao suporte dos requisitos enumerados na Seção 2.3, os parágrafos seguintes apresentam uma breve a avaliação do *framework*:

- **Conectividade em plataformas heterogêneas.** As atuais tecnologias de *middeware* apresentam muitas soluções interessantes para conectividade. Em relação a estas soluções, esta tese apresenta abordagens diferentes relacionadas a flexibilidade e simplicidade para interconexão de componentes e para a sincronização envolvendo adaptações dos mecanismos de suporte a interações.

- **Abordagem arquitetural.** De uma forma geral, o comentário que merece destaque com relação à abordagem arquitetural do Cosmos está relacionado ao modelo de metacomponentes definido para dar suporte aos mecanismos de descrição de aplicações e de reflexividade.

- **Adaptação e QoS.** Com relação ao requisito de adaptação, a maioria dos *frameworks* de componentes abordam a questão da adaptação no domínio de aplicações corporativas, onde praticamente não existe restrições rígidas com relação à QoS. Neste contexto, a abordagem do Cosmos oferece soluções interessantes para gerenciamento de adaptação e QoS de componentes na camada do *middleware*.

- **Configuração e reconfiguração.** O Cosmos apresenta níveis elevados de flexibilidade para tratamento de configuração e reconfiguração dinâmicas. Este fato deve-se à abordagem definida para configuração com base no modelo de metacomponentes.

6.2.1 Análise Comparativa

Um fator de diferença do Cosmos em relação ao PREMO [7], A/V *Streams* [8] e o NMM [11] é a adoção do conceito de componentes, o que provoca uma mudança em relação às visões de desenvolvimento de sistemas e gerenciamento de configuração. Ao incorporar o modelo de desenvolvimento baseado em componentes, pode-se realizar com relativa facilidade ajustes evolucionais no próprio *middleware*, trocando, por exemplo, alguns dos seus componentes.

O *framework* Cosmos utiliza o conceito de propriedades tomando como base as propostas PREMO [7] e A/V *Streams* [8]; no entanto, o uso do conceito de propriedades no Cosmos é bem mais simples devido também à adoção do conceito de componentes. Componentes limitam a visibilidade de seus comportamentos internos, de modo que no Cosmos as propriedades são usadas muito mais intensamente para fins de negociação e seleção de componentes. Para a definição do comportamento de uma aplicação, a noção de propriedades se resume à seleção de parâmetros de operação do componente.

Outro ponto que merece destaque é o modelo de componentes do Cosmos. Este modelo provê a base do *framework* para a construção de componentes arquiteturais genéricos (componentes com níveis elevados de abstração) e configuráveis. Estes componentes são definidos no Cosmos como abstratos, podendo posteriormente serem estendidos e configurados de acordo com os requisitos estabelecidos pela plataforma onde o *middleware* será instalado e instanciado. O modelo de configuração definido no Cosmos é neutro em relação ao uso de linguagens específicas, podendo ser utilizado junto com técnicas de computação reflexiva para dar suporte à integração de uma diversidade de componentes. Desta forma, componentes poderão potencialmente vir a ser desenvolvidos usando diferentes tecnologias, desde que obedeçam as regras prescritas pelo modelo de componentes Cosmos. Um aspecto importante do Cosmos em relação à maioria das soluções baseadas em componentes é a abordagem para gerenciamento de QoS.

A abordagem de manter as definições de metadados em um único domínio – o domínio onde reside o componente *Configurator* mestre, é uma simplificação do modelo em relação às abordagens citadas no Capítulo 3. Deve-se destacar que esta definição não implica em perda de generalidade e flexibilidade. Adicionalmente, esta definição tem uma forte motivação que justifica a decisão. Conforme mencionado na introdução, um dos objetivos deste trabalho foi desenvolver soluções para o domínio de sistemas de TVDI. Deve-se destacar que a maioria das aplicações para estes sistemas é executada em contextos locais em *set-top-boxes*, provavelmente sem canal de interação. Outra questão que também contribuiu para esta opção decorre do fato de que o elemento determinante que vai caracterizar o nível de suporte à execução de aplicações e à respectiva QoS, é a capacidade da plataforma correspondente ao terminal de acesso ao sistema. Ou seja, este sistema conhece profundamente suas capacidades. Desta forma, esta plataforma é o lugar ideal para manter as descrições dos componentes da aplicação, de onde serão comandadas as negociações para realizar a configuração da aplicação. Este modelo também é relevante para alguns tipos de aplicações no

contexto Web, onde parte das condições de negociação poderá estar limitada pela capacidade da plataforma local onde a aplicação será executada. Adicionalmente, mantendo as descrições e alterações concentradas sob a coordenação do *Configurator* mestre, tem-se a garantia da manutenção da consistência do sistema. Esta simplificação tem como implicação tornar o *Configurator* num ponto central de falha. Para o cenário de sistemas de TVDI sem canal de interação esta limitação não tem impactos.

Considerando o modelo de gerenciamento de QoS definido para o Cosmos, outros fatores de diferenciação em relação às propostas relacionadas no Capítulo 3 correspondem ao conceito de faixas de valores de QoS e ao gerenciamento de mudança de faixa de acordo com as limitações e condições temporárias dos recursos da plataforma. O modelo foi incorporado no *middleware* AdapTV e utilizado para especificação de requisitos de QoS e definição de políticas de adaptação na aplicação exemplo. Na implementação foram usados monitores de QoS para observar as condições de operação do canal de comunicação. Vale lembrar o que foi anteriormente mencionado: no Cosmos, um monitor de QoS tem um papel estritamente limitado, consistindo em observar e notificar possíveis ocorrências de violações de QoS. A adaptação é sugerida por um componente denominado gerente de QoS, cabendo ao *Configurator* acatar ou não a sugestão de adaptação.

No que se relaciona ao modelo de interconexão de componentes, a idéia inicial foi definir um componente inspirado no conceito de conexão virtual do modelo PREMO. Embora este conceito tenha sido herdado, a abordagem definida no Cosmos simplifica a integração, uso e gerenciamento dos componentes envolvidos no modelo.

O conceito de portas, sob a visão do componente recurso virtual é bem mais simples no Cosmos. A simplicidade deve-se ao fato de que, para o componente basta que ele conheça a interface operacional das portas que ele precisa utilizar. Questões de configuração e negociação de propriedades são totalmente abstraídas do componente e da aplicação, uma vez que as mesmas são realizadas no contexto do *middleware* através do componente *Configurator*.

O modelo dá suporte a várias topologias de interconexão envolvendo, inclusive, a possibilidade de suportar simultaneamente o uso de diferentes tecnologias; adicionalmente, deve-se destacar a capacidade do modelo para realizar adaptações dinâmicas, onde são definidas técnicas como clonagem e uso simultâneo e temporário de canais durante a realização e transição de estados em uma adaptação. Este modelo é bem mais simples do que a técnica denominada *paralel binding* [12] que foi adotada no

middleware NMM [11] [12]. O *middleware* NMM também usa em sua concepção os conceitos de conexão virtual do PREMO.

A arquitetura de interconexão é genérica, possibilitando a integração de uma variedade de tipos de componentes em ambientes heterogêneos e distribuídos. A API proposta é relativamente simples e bastante flexível, cujas operações definidas dão suporte aos principais componentes definidos no *middleware* AdapTV e aos conceitos apresentados pelo *framework* Cosmos.

O modelo de interconexão proposto e a arquitetura de implementação definida se constituem, na nossa visão, em contribuições importantes. Usando este modelo, novas aplicações poderão ser construídas integrando, por exemplo, tecnologias como *Internet* e TVDI através da composição de *pipes* envolvendo componentes intermediadores que poderão ser interconectados através de componentes *VirtualConnection*. A avaliação deste modelo ocorreu de forma integrada com a implementação do protótipo do *middleware* AdapTV realizada, envolvendo além do modelo de interconexão, os modelos de metacomponentes, configuração e suporte à adaptação dinâmica baseada em QoS. Esta implementação produziu bons resultados, criando expectativas positivas em relação à evolução das idéias propostas.

O escopo da implementação se limitou aos componentes da camada de configuração e gerenciamento de recursos do *middleware* AdapTV e a uma aplicação-exemplo que foi executada no protótipo. Para o desenvolvimento da aplicação foram utilizados os modelos de especificação, configuração, gerenciamento de QoS e de interconexão de componentes definidos, ficando assim demonstrada experimentalmente a potencialidade e a viabilidade de implementação do Cosmos. Deve-se destacar que o resultado deste esforço se constitui, na nossa visão, numa contribuição importante e inovadora, fornecendo uma nova visão em relação ao gerenciamento de configuração para sistemas distribuídos multimídia. Explorando esta visão, o gerenciamento da diversidade de recursos, sistemas operacionais e dispositivos fica bem mais simples nestes sistemas.

6.3 Trabalhos Futuros

Considerando a abrangência do *framework*, várias questões não puderam ser tratadas na tese. Neste sentido, são enumeradas a seguir algumas proposições que darão maior visibilidade ao *framework*, as quais esperamos tratar no contexto de projetos futuros.

Definição de ferramentas de configuração visuais

Uma idéia interessante consiste em definir uma ferramenta de configuração interativa, com uso de componentes visuais para identificação de componentes, relacionamento de dependências, configuração de propriedades, ligação de portas, composição de grupos e requisitos de reconfiguração dinâmica pré-programada e não-programada. Esta ferramenta pode manter um repositório envolvendo várias configurações previamente construídas (representadas como grafos de metacomponentes). O reuso de configurações poderia ser incorporado no modelo também de forma interativa. O desafio neste caso é tratar a consistência das operações. Uma possível delimitação para tornar viável uma implementação simplificada seria eleger alguns casos de uso e explorar estes cenários.

Definição de um modelo de eventos e sincronização

O suporte ao tratamento de eventos e à sincronização são de suma importância para aplicações multimídia, em especial quando estão envolvidos componentes distribuídos. Estas questões foram pouco trabalhadas na tese uma vez que para as necessidades do protótipo desenvolvido como prova de conceito, as APIs de Java foram suficientes. No entanto, estas questões merecem uma atenção especial, devendo portanto ser tratadas no escopo de futuros projetos.

Outra questão que deve ser tratada é a validação do modelo de adaptação envolvendo troca dinâmica de componentes. O modelo de adaptação implementado envolveu a troca de propriedades e a questão da sincronização foi resolvida com a clonagem de canais. Adaptações envolvendo a troca de componentes ampliam a complexidade, o que pode levar a necessidade de novas técnicas, como por exemplo, a clonagem do componente *ApplicationProxy*.

Avaliação e definição de políticas de adaptação envolvendo mudanças de canais.

As experiências realizadas na implementação do protótipo foram direcionadas para o trabalho de integração dos vários modelos. A aplicação definida foi bastante simples e teve como objetivo mostrar o potencial do modelo de adaptação. Um trabalho que pode ser desenvolvido consiste em identificar padrões de comportamento que estabeleçam condições preliminares de adaptação. Para isto, seria necessário fazer muitas observações de modo a analisar os dados resultantes e montar um

conjunto de estratégias a serem adotadas. Ainda relacionado com o componente *VirtualConnection*, deve-se realizar testes envolvendo comunicação *multicast* integrando inclusive diferentes tecnologias simultaneamente.

Validar os componentes do AdapTV em uma plataforma real de TVDI

As experiências envolvidas com mudança dinâmica de propriedades de fluxos no AdapTV sinalizaram que o modelo é eficaz no gerenciamento de variações de QoS. Desta forma, uma experiência interessante é migrar os componentes do AdapTV para uma plataforma real de TVDI.

Avaliação de desempenho da arquitetura

Outro trabalho futuro proposto é a avaliação de desempenho da arquitetura e dos componentes associados à implementação do *framework*. Nesta avaliação, devem ser considerados os retardos associados por exemplo a tempo de troca de formato, tempo de troca de canal, ou mesmo de componentes no contexto de uma implementação do *framework*.

Referências Bibliográficas

[1] S. J. Gibbs and D. C. Tsichritzis. *Multimedia Programming – Objects, Environments and Frameworks*. Adson-Wesley, 1995.

[2] R. Orfali and D. Harkey. *Client/Server Programming with Java and CORBA*. 2nd Edition Wiley, 1998.

[3] W. Emmerich. *Engineering Distributed Object*. Wiley, 2000.

[4] Object Management Group. Common Object Request Broker Architecture and Specification. Technical Report/2002-05-08, December 2002. `http://www.omg.org/`.

[5] D. Box. *Essential COM*. Addison-Wesley, 1998.

[6] A. Wollrath, R. Riggs and J. Waldo. A Distributed Object Model for the Java System. In *USENIX Computing Systems, 9(4),* November/December 1996.

[7] D. Duke, I. Herman. A Standard for Mulimtedia Middleware. In *Proceedings of ACM International Conference on Multimedia*, 1998.

[8] Object Management Group. The Audio/Video Streams Specification – V1.0, Tech. Rep. formal/2000-01-03. `http://www.omg.org`.

[9] R.Vanegas, J. A. Zinky, J. P. Loyall, D. A. Karr, R. E. Schantz and D. E. Bakken. Quo's runtime support for quality of service in distributed objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, England, September 1998.

[10] P.K. McKinley, S. M. Sadjadi and E. P. Kasten. Composing Adaptive Software. In *IEEE Computer Society*, 2004.

[11] M. Lohse, M. Repplinger, P. Slusallek. An Open Middleware Architecture for Network-Integrated Multimedia. In *IDMS/PROMS'2002*, Coimbra, Portugal, 2002.

[12] Lohse, M. Network-Integrated Multimedia Middleware Services and Applications. PhD These. Universitat des Saarlands.Saarbrucken, Germany, 2005.

[13] ISO-IEC. Open Distributed Processing Reference Model, Part 1: Overview. International Standard ISO/IEC 10746-1, 1995.

[14] Y.Inoue, D.Cuha and H. Berndt, The TINA Consortium. *Communications Magazine, IEEE*. Sep 1998 V. 36, Issue: 9. p.130-136.

[15] H. A. Duran-Limon and G. S. Blair. QoS management specification support for multimedia middleware. *The Journal of Systems and Software* 72 (2004), p. 1–23. Elsevier.

[16] D. C. Schmidt. ACE: an Object-Oriented Framework for Developing Distributed Applications. In *Proceedings of the USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, Apr. 1994.

[17] Center for Distributed Object Computing. The ADAPTIVE Communication Environment (ACE). Washington University. `http://www.cs.wustl.edu/ schmidt/ACE.html`.

[18] G. Blair, A.Coulson, A. Andersen, L. Blair, M. Clarke, F. M. Costa, M. Duran, N. Parlavantzas and K. Sailsoki. A principled approach to supporting adaptation in distributed mobile environments. In *International Symposium on Software Engineering for Parallel and Distributed Systems - PDSE´200*. 2000.

[19] E. G. Guimarães. Um Modelo de componentes para aplicações telemáticas e ubíquas. Tese de Doutorado. Faculdade de Engenharia Elétrica e de Computação. Unicamp.2004.

[20] G. S. Blair et al. The Design and Implementation of Open ORB Version 2. *In IEEE Distributed Journal,* Vol 2, N. 6,2001.

[21] M. Völter, A. Schmid and E. Wolff. *Server Component Patterns Component Infrastructures Illustrated with EJB*. Wiley Series in Software Design Patterns. Wiley and Sons 2002.

[22] B. Councill, . and G. T. Heineman. *Definition of a Software Component and Its Elements*. *Component-Based Software Engineering: Putting the Pieces Together*. Ed. Addison-Wesley. 2001.

[23] J. B. Postel. Transmission control protocol', RFC 793, IETF, 1981.

[24] R. Neigenmann and D. Lilja. Von Neuman Computers. `http:dynamo.ecn.purdue.edu/~eigenman/reports/vN.pdf`.

[25] P K. McKinley, S. M. Sadjadi, E. P. Kasten and B. Cheng. A Taxonomy of Compositional Adaptation. T.R.MSU-CSE-04-17. Michigan State University. July 2004.

[26] L. Bergmans and M. Aksit. Aspects and crosscutting in layered middleware systems. In *Proceedings of the IFIP/ACM (Middleware2000), Workshop on Reflective Middleware Palisades*, NY, Apr 2000, p. 23--25.

[27] C. Szyperski. *Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley, 2002.

[28] C. Szyperski, Component Technology. What, Where, and How?. *ICSE, p. 684 25th International Conference on Software Engineering (ICSE'03)*, 2003.

[29] F. M. Costa and G. S. Blair. Integrating Meta-Information Management and Reflection. In *Middleware(2000) International Symposium on Distributed Objects and Applications*. 2000.

[30] W. Emmerich. Distributed component technologies and their software engineering implications. *International Conference on Software Engineering. Proceedings of the 24th International Conference on Software Engineering*. P 537-546,. ACM Press New York, NY, USA,2002.

[31] G. Elias, A. Lopes, F. Borelli and M. F. Magalhães. Exploring an Open, Distributed Multimedia Framework to Design and Develop an Adaptive Middleware for Interactive Digital Television Systems. In *19th ACM Symposium on Applied Computing (SAC),* Nicósia, Chipre, 2004.

[32] M. Hayden. The Ensemble System. PhD, TR98-1662, Cornell University, January, 1998.

[33] Microsoft Corporation. The Common Language Runtime (CLR). `http://msdn.microsoft.com/netframework/programming/clr/`

[34] IONA. *ORBIX 2: distributed object technology*. Dublin, IONA Technologies, 1995.

[35] ORBacus. ORBacus For C++ and Java. Object Oriented Concepts, Inc.2000.

[36] R. Klefstad, D. C. Schmidt and C. O'Ryan. Towards highly configurable real-time object request brokers. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, April - May 2002.

[37] R. Baldoni, C. Marchetti and A. Termini. Active software replication through a three-tier approach. In *Proceedings of the 22th IEEE International Symposium on Reliable Distributed Systems (SRDS02)*, Osaka, Japan, pp. 109–118, October 2002.

[38] S. M. Sadjadi and P. K. McKinley. ACT: An adaptive CORBA template to support unanticipated adaptation. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS'04),* Tokyo, Japan, March 2004.

[39] G. Kiczales, J. des Rivieres and D. G. Bobrow. *The Art of Metaobject Protocols*. MIT Press, 1991.

[40] Python. `http://www.python.org/`

[41] M. Tatsubori, S. Chiba, K. Itano and M. O. Killijian. OpenJava: A class-based macro system for Java. In *Proceedings of OORaSE*, pp. 117–133, 1999.

[42] J. O. Guimarães. Reflection for statically typed languages. In *Proceedings of 12th European Conference on Object-Oriented Programming (ECOOP'98),* pp. 440–461, 1998.

[43] V. Adve, V. V. Lam and B. Ensink. Language and compiler support for adaptive distributed applications. In *Proceedings of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM2001),* Snowbird, Utah, June 2001.

[44] E. P. Kasten, P. K. McKinley, S. M. Sadjadi and R. E. K. Stirewalt. Separating introspection and intercession in metamorphic distributed systems. In *Proceedings of the IEEE Workshop on Aspect-Oriented Programming for Distributed Computing (with ICDCS'02),* Vienna, Austria, pp. 465–472, July 2002.

[45] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, vol. 2072, pp. 327–355, 2001.

[46] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of ACM*, pp. 51–57, October 2001.

[47] S. M. Sadjadi, P. K. McKinley, R. E. K. Stirewalt and B. H. Cheng. TRAP: Transparent reflective aspect programming. Tech. Rep. MSU-CSE-03-31, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, November 2003.

[48] R. F. G. Cerqueira. Um Modelo de Composição Dinâmica entre Sistemas de Componentes de Software. Tese de Doutorado. Departamento de Informática, PUC-Rio de Janeiro. 2000.

[49] H. A. Duran-Limon and G. S. Blair. QoS management specification support for multimedia middleware. *The Journal of Systems and Software* 72 (2004) 1–23.

[50] O. Layaıda and D. Hagimont. Adaptive video streaming for embedded devices. In *Software Engineering, IEE Proceedings*. V.152, Issue 5, p: 238- 244, 7 Oct. 2005.

[51] V. Issarny, V., C. Bidan. Aster: A Framework for Sound Customization of Distributed Runtime Systems. *In Proceedings of the 16th IEEE International Conference on Distributed Computing Systems (ICDCS'96).* May 1996.

[52] G. Blair, L. Blair, V. Issarny, P. Tuma, A. Zarras. The role of software architecture in constraining adaptation in component-based middleware platforms. *In Proceedings of Middleware'00 -- The ACM/IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, 2000.

[53] M. Hemi, U. Hengartner, P. Steenkiste and T. Gross. MPEG system Streams in Best Effort Networks. In *Proc. of PacketVideo' 99*, Cagliari, Italy, April 1999.

[54] J. Vass, S. Zhuang, J. Yao and X. Zhuang. Efficient mobile video access in wireless environments. In *IEEE Wireless Communications and Networking Conference*, New Orleans, LA, Sept. 1999.

[55] M. Johnson. An RTP to HTTP video gateway. In *Proc of the World Wide Web Conference* 2001.

[56] Z. Lei and N.D. Georganas. H.263 Video Transcoding for Spatial Resolution Downscaling, In *Proc. of IEEE International Conference on Information Technology: Coding and Computing (ITCC) 2002*, Las Vegas, April 2002.

[57] S. McCanne and V. Jacobson. VIC: A flexible framework for packet video. In *Proc. of ACM Multimedia' 95,* November 1995.

[58] V. Hardman, A. Sasse, M. Handley and A. Watson. Reliable Audio for Use over the Internet. In *Proc. of INET' 95, June 1995*, Honolulu, Hawai.

[59] H. Schulzrinne et al. RTP: A Transport Protocol for Real- Time Applications. RFC 1989.

[60] B. Li and K. Nahrstedt. A Control-based Middleware Framework for Quality of Service Adaptations. In *IEEE Journal on Selected Areas in Communications*, Vol. 17, No. 9, September 1999.

[61] E. Amir, S. McCanne and Z. Hui. An Active Service Framework and its Application to Real-time Multimedia Transcoding. *ACM Computer Communication Review*, vol. 28, no. 4, September, 1998.

[62] A. P. Black, J. Huang, R. Koster, J. Walpole and C. Pu. Infopipes: an Abstraction for Multimedia Streaming. In *Multimedia Systems (special issue on Multimedia Middleware, 2002).*

[63] X. Fu, W. Shi, A. Akkerman and V. Karamcheti. CANS: Composable, adaptive network services. *USENIX Symposium on Internet Technologies and Systems (USITS),* March 2001.

[64] Z. Morley Mao, H. W. So, B. Kang and R.H. Katz. Network Support for Mobile Multimedia using a Self-adaptive Distributed Proxy. In *11th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV-2001).*

[65] N. Rodriguez, R. Ierusalimschy and R. Cerqueira. R. Dynamic configuration with CORBA components. In *Proceedings of Fourth International Configurable Distributed Systems Conference* 4-6 May 1998 Page(s):27 – 34.

[66] R. Ierusalimschy, L. Figueiredo and W. Celes. Lua – an extensible extension language. *Software: Practice and Experience*, 26(6), 1996.

[67] E. G. Guimarães, A. Lopes, J. A. Coello and M. F. Magalhães. A Distribuição do Ambiente para Desenvolvimento de Software de Tempo Real STER. In *Seminário Franco Brasileiro em Sistemas Informáticos e Distribuídos*, Florianópolis, SC, Setembro 1989.

[68] A. B. Lopes. LPM e LCM: Linguagens para Programação e Configuração de Módulos. Programa de Mestrado em Sistemas e Computação, Campina Grande, PB, 1987.

[69] N. Medvidovic, R. N. Taylor, A Classification and Comparison Framework for Software Architecture Description Languages. *In IEEE Transactions on Software Engineering, Vol. 26, No. 1, pp. 70-96*, January 2000.

[70] J. Magee, N. Dulay, S. Eisenback and J. Kramer. Specifying Distributed Software Architectures. *Proceedings of the Fifth European Software Engineering Conference*, Sitges, Spain, September, 1995.

[71] J. Magee, N. Dulay and J. Kramer. Regis: A Constructive Development Environment for Distributed Programs. In *IEE/IOP/BCS Distributed Systems Engineering*, 1(5), pp. 304-312, September 1994.

[72] R. Helm, I. M. Holland and D. Gangopadhyay. Contracts: Specifying Behavioural Compositions in Object-Oriented Systems. In *ECOOP/OOPSLA'90 Proceedings*, October 1990.

[73] D. Garlan, R. Allen and J. Ockerbloom. Architectural Mismatch or Why it's hard to build systems out of existing parts. In Proceedings of the Seventeenth International Conference on Software Engineering, SeattleWA, April 1995.

[74] I. Crnkovic. Component-based Software Engineering – New Challenges in Software Development. `http://www.mrtc.mdh.se/publications/0328.pdf`.

[75] A. Lopes, F. Borelli, G. Elias, and M. Magalhães. A Component-Based Configuration Framework for Open, Distributed Multimedia Systems. In *18th IEEE International Conference on Advanced Information Networking and Applications (AINA),* Fukuoka, Japan, 2004.

[76] F. Borelli, A. Lopes and G. Elias. A XML-Based Component Specification Model for an Adaptive Middleware of Interactive Digital Television Systems. In *18th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, Fukuoka, Japan, 2004.

[77] A. Lopes, F. Borelli, G. Elias, and G. Lemos. Projeto e Implementação de um Middleware para sistemas de Televisão Digital Interativa. In *Webmidia 2004*.

[78] F. Borelli. BRICKS: Um Modelo de Componentes com Suporte à Composição Baseada em Negociação de Propriedades de Interfaces. Dissertação de Mestrado, Programa de Pós-Graduação em Sistemas e Computação, UFRN-Natal, 2006.

[79] R. E. Johnson. Component, Frameworks, Patterns. In *Proc. Simposium on Software Reusability*, pp.10-17, May 1997.

[80] M. Fayad, D. Schmidt and R. Johnson. *Component-Framework Systems*. Wiley & Sons, New York City, 2000.

[81] A. B. Lopes, G. Elias and M. F. Magalhães, COSMOS: Um Framework para Configuração e Gerenciamento de Sistemas Multimídia Distribuídos Abertos. In *III Workshop de Teses e Dissertações em Multimídia, Hipermídia e Web*. Webmidia 2003, Salvador,BA.

[82] A. B. Lopes, F. Borelli, G. Elias, G., Lemos and M. Magalhães. Uma Arquitetura para Configuração e Gerenciamento de Recursos em um Middleware para Sistemas de Televisão Digital Interativa. In *XXI Simpósio Brasileiro de Telecomunicações*, 2004, Belém, PA.

[83] F. Amaro, F. Lopes, C. Silva, D. Oliveira, A. Lopes, G. Elias and G. L. Souza. Especificação e Gerenciamento de QoS em um Middleware paraSistemas de Televisão Digital Interativa. In *Webmidia 2005*.

[84] A. Lopes, F. Amaro, G. Elias, G. Lemos and M. Magalhães. QoS Specification and Management in a Middleware for Distributed Multimedia Systems. In *20th IEEE AINA,* Vienna*,* Austria, 2006.

[85] A. Lopes, C. E. Silva, F. Borelli, G. Elias, and G. Lemos. Uma Arquitetura de Interconexão de Componentes para Sistemas Multimídia Distribuídos. In *4° WDBC, João Pessoa, PB, 2004*.

[86] C. Silva, D. Oliveira, F. Lopes, F. Amaro, A. Lopes, G. Elias and G. Lemos. Um Modelo de Interconexão de Componentes e sua Implementação em um Middleware para para Sistemas de Televisão Digital Interativa. In *XXII Simpósio Brasileiro de Telecomunicações*, 2005, Campinas, SP.

[87] Silva C, Lopes, A. Elias G., Lemos G. and Magalhaes M., "A Component Interconnection Model for Interactive Digital Television". In 20th IEEE AINA, Vienna, Austria, 2006.

[88] L. E. C. Leite, C. E. Batista, G. L. Souza Filho, R. Kulesza, L. G. P. Alves, G. Bressan, R. F. Rodrigues and L. F. G. Soares. FlexTV - Uma Proposta de Arquitetura de Middleware para o Sistema Brasileiro de TV Digital. In *Revista de Engenharia de Computação e Sistemas Digitais*, v. 2, p. 29-50, 2005.

[89] F. Kon, F. Costa, G. Blair and R. H. Campbell. The Case for Reflective Middleware. In *Communications ACM,* vol. 45, pp. 33-38, June 2002.

[90] R. C. M. Medeiros, A. B. Lopes and M. F. Magalhães. Proposta de um Ciclo de Vida estendido para o Modelo ISO/PREMO e sua Implementação em Ambiente CORBA. In *SBMIDIA*, p 87-98. Rio de Janeiro,1998.

[91] C. Ribeiro, G. L. Souza and A. B. Lopes. Um Modelo para Especificação Formal da Configuração de Aplicações Baseadas no Premo. (Resumo). In *17^0 Simpósio Brasileiro de Redes de Computadores.* p. 371-372 Salvador, BA, 1999.

[92] C. Ribeiro, G. L. Souza and A. B. Lopes. Especificação Formal de um Modelo de Sincronização Baseado no PREMO. In *SBMIDIA´99*. p 82-101, Goiânia, 1999.

[93] A. B. Lopes, M. F. Magalhães, L. Desiderá L. and N. P. Tizzo. Uma Plataforma para Programação Multimídia Baseada no Modelo ISO/PREMO e sua Implementação. *In XXIII CLEI, Conferência Latino Americana de Informática*, Valparaiso – Chile,1997. v.2, p 705-714

[94] J. Cheesman, J. Daniels. *UML Components A Simple Process for Specifying Component-Based Software.* Addison-Wesley, 2001.

[95] G. Blakowsky and R. Steinmetz. A Media Synchronization Survey: Reference Model, Specification, and Case Studies. In *IEEE Journal on Slected Areas in Communications*, vol 14, No 1, January 1996.

[96] T. Phan, G. Zorpas and R. Bagrodia. An extensible and scalable content adaptation pipeline architecture to support heterogeneous clients. In *22nd International Conference on Distributed Computing Systems (ICDCS 2002)*.

[97] C. Bouras, A. Gikamas, A. Karaliotas and K. Stamos. Architecture and Performance Evaluation for Redundant Multicast Transmission Supporting Adaptive QoS. *Multimedia Tools and Application*, V.25, pp 85-110, Springer Science Business Media, 2005.

[98] C. Diot. On QoS and traffic engineering and SLS-related Work by Sprint. In *Worshop on Internet Design for SLS Delivery*,Amesterdam, The Netherlands, 25-26 jan, 2001.

[99] J. Pandhye, J. Kurose, D. Towsley and R. Koodli. A model based TCP-friendly rate control protocol. In Proc. In *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV),* Basking Ridge, NJ June 1999.

[100] H. A. Duran-Limon. A Resource Management Framework for Reflective Multimedia Middleware. PhD dissertation, University of Lancaster, 2001.

[101] R. Koster. A Middleware Platform for Information Flows. PhD These, Universitat Kaiserslautern, Kaiserslautern, 2002.

[102] P. Shenoy, S. Hasan, P Kulkam, and K. Ramamriham. Middleware versus Native OS Support: Architectural Considerations for Supporting Multimedia Applications. In *Real-Time and Embedded Technology and Applications Symposium*, pp. 23-32, 2002.

[103] C. M. F. Ribeiro. ESCHER: Uma Arquitetura de Qualidade de Serviço para Tratar a Percepção do Usuário. Tese de Doutorado. Programa de Pós-Graduação, CIN-UFPE, 2004.

[104] MPEG standards. `http://www.chiariglione.org/mpeg/standards.htm`.

[105] D. Birrel and B. J. Nelson. Implementing remote procedure calls. In *ACM Transactions on Computer Systems (TOCS)* Volume 2 , Issue 1 Pages: 39 – 59. February 1984.

[106] M. Pesce. *Programming Microsoft DirectShow for Digital Video and Television*. Microsoft Press. 2003.

[107] N. Wang, D. Schmidt, A. Gokhale, C.D. Gill, B; Natarajan, C. Rodrigues, J.P. Loyall and R. Schantz. Total Quality of Service Provisioning in Middleware and Applications. In *Microprocessors and Microsystems, special issue on Middleware Solutions for QoS-enabled Multimedia Provisioning over the Internet*, Paolo Bellavista ed., vol. 27, no. 2, pp. 45-54, March 2003

[108] F. Eliassen et al. Next Generation Middleware: Requirements, Architecture, and Prototypes. In *The Seventh IEEE Workshop on Future Trends of Distributed Computer Systems*. 1999.

[109] C. Zhang and H. A. Jacobsen. Refactoring Middleware with Aspects. In *IEEE Transactions on Paralel and Distributed Systems.* V.14, No.11, pp. 1058-1073, Nov 2003.

[110] OMG. CORBA Components, OMG formal/2006-04-01.

[111] Microsoft. .NET Web Services Platform. http://www.microsoft.com/net.

[112] Sun Microsystems. Enterprise JavaBeans specification.
http://java.sun.com/products/ejb/

[113] Sun Microsystems. Java Platform Enterprise Edition 5.
http://java.sun.com/javaee/technologies/javaee5.jsp

[114] JBoss .J2EE application server. Htpp:// www.jboss.org/

[115] K. Balasubramanian, N. Wang and D. C. Schmidt. Towards Composable Distributed Real-time and Embedded Software. In *Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03),* 2003.

[116] R. E. Schantz and D. C. Schmidt. *Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications*. Encyclopedia of Software Engineering (J. Marciniakand G. Telecki, eds.), New York: Wiley & Sons, 2002.

[117] OMG . RT CORBA 1.1 specification.
http://www.omg.org/technology/documents/spec_catalog.htm

[118] D. C. Schmidt, D. L. Levine and S. Mungee. The Design and Performance of Real-Time Object Request Brokers. In Computer Communications 21 (4), pp. 294–324, 1998.

[119] G. Coulouris, J. Dollimore and T. Kindberg. Distributed Systems:Concepts and Design. Addison-Wesley, third edition, 2001.

[120] M. Shaw and D. Garlan. Sofware Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, 1996

[121] M . M. Rakic, N.Mehta and N. Medivdovic. Architectural Style Requirements for Self-Healing Systems. In Proceedings of the First ACM Workshop on Self-Healing Systems. Charleston, 2002.

[122] N. Mehta. Distilling Software Architectural Primitives from Architectural Styles. TR USC-CSE-2002-504, USC, 2002.

[123] S. Cheng et. al. Using Architectural Style as a Basis for Self-repair. In *Proceedings of the 2002 IEEE/IFIP Conference on Software Architectures (WICSA 2002)*, Montreal, Canada, August 25-30, 2002.

[124] P. Oreizy et. al. An Architecture-Based Approach to Self-Adaptive Software. In *IEEE Intelligent Systems*, Vol. 14, No. 3, May/June 1999.

[125] G. S. Blair, G. Coulson and P. Grace. Research directions in reflective middleware: the Lancaster experience. In *Proceedings of the 3rd Workshop on Reflective and Adaptive Middleware (RM2004),* co-located with Middleware 2004, Toronto, Ontario, Canada, October 2004.

[126] T. Fitzpatrick, G. S. Blair, G. Cuolson, N. Davies and P. Robin. Supporting Adaptive Multimedia Applications through Open Bindings. In Proc. ICCDS´98, Annapolis, MD, USA,1998.

[127] E. Gamma, R. Helm, R. Johnson and J. Vlissides. DesignPatterns. Addision-Wesley, 1994.

[128] D. J. Duke, I. Herman and M. S. Marshall. PREMO: A Framework for Multimedia Middleware – Specification, Rationale*, and Java Binding*. Lecture Notes in Computer Science; vol. 1591. Springer, 1999.

[129] M. Repplinger, F. Winter, M. Lohse and P. Slusallek. Parallel Bindings in Distributed Multimedia Systems. In *icdcsw, pp. 714-720, Seventh International Workshop on Multimedia Network Systems and Applications (MNSA) (ICDCSW'05),* 2005.

[130] T. Fitzpatrick. Open Componente-Oriented Multimedia Middleware for Adaptive Distributed Applications. PHD Thesis. Computing Department, Lancaster University, 1999.

[131] D.C. Schmidt, A.S. Gokhale, T.H. Harrison, G. Parulka., A High-Performance End System Architecture for Real-Time CORBA. *In IEEE Communications Magazine, Vol. 35, No. 2, Feb. 1997.*

[132] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L.C. Magalhães, and R. Campbell. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In Proc. of the International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware), pp 121–143, 2000.

[133] K. Nahrstedt, D. Xu, D. Wichadakul and B. Li. QoS-Aware Middleware for Ubiquitous and Heterogeneous Environments. In *Communications Magazine, IEEE.* V. 39, Issue 11, p: 140-148, 2001

[134] F. Costa and G. S. Blair. Integrating Meta-Information Management and Reflection in Middleware. In Proc. of the Int´l Symposium on Distributed Objects and Applications (DOA´00), IEEE Press, Piscataway, N.J, Sep. 2000.

[135] Multimedia Home Platform. `http://www.mhp.org`.

[136] DTV Application Software Environment. `http://www.dase.org`.

[137] Advanced Common Application Platform (ACAP). ATSC ProposedStandard, fevereiro, 2004. `http://www.atsc.org/standards/cs_documents/cs_101a.pdf`.

[138] Association of Radio Industries and Business. `http://www.arib.or.jp`.

[139] OpenCable Specifications. `http://www.opencable.com/specifications`.

[140] JavaTV. Java Technology in Digital TV. `http://java.sun.com/products/javatv/`.

[141] JMF. Java Media Framework (JMF). `http://java.sun.com/products/jmf/`.

[142] DAVIC 1.4 Part 2. DAVIC Specification Reference Models and Scenarios,1998. http://www.havi.org.

[143] HAVi Level 2 Graphical User-Interface. Specification of the Home Audio/VideoInteroperability (HAVi) Architecture. HAVi. http://www.havi.org.

[144] Globally Executable MHP (GEM). ETSI Standard, May 2004.

[145] Advanced Television System Committee. http://www.atsc.org.

[146] L. F. G. Soares and R. F. Rodrigues. Nested Context Model 3.0: Part 5 – NCL (Nested Context Language). Relatório Técnico de Pesquisa da série de Monografias do Departamento de Informática da PUC-Rio. 2005.

[147] G. Elias. SOS: A Framework for Distribution, Management and Evolution of Component-Based Software Systems over Open Networks. Tese de Doutorado em Ciência da Computação. Centro de Informática. UFPE, 2002.

[148] A. Corsaro, D. Schmidt, R. Klefstad and C. O'Ryan.Virtual Component a design pattern for memory constrained embedded applications. In *Proc. of the 9th Conference on Pattern Language of Programs (PLoP 2002).* Monticello, Illinois, USA. September, 2002.

[149] R. Allen, R. Douence and D. Garlan. *Specifying and Analyzing Dynamic Software Architectures*. In Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98), volume 1382 of Lecture Notes in Computer Science, pages 21--37. Springer-Verlag, 1998.

[150] OMG. Reusable Asset Specification (RAS) V.2.2, formal/2005-11-02.

[151] R. Moreira, G. Blair and E. Carrapatoso. FORMAware: Framework of Reflective Components for Managing Architecture Adaptation. *In 3th Int. Worshop on Software and Middleware.* Orlando, May 2002.

[152] R. Moreira, G. Blair, E. Carrapatoso. Constraint Architectural Reflection for Safely Managing Adaptation. *In 2nd Workshop on Reflective and Adaptive Middleware in Middleware 2003 International Middleware Conference*. Rio de Janeiro, June 2003.

[153] H. A. Duran-Limón. A Resource Management Framework for Reflective Multimedia Middleware. PHD Thesis. Computing Department, Lancaster University, 2001.

[154] Costa F. e Blair G. Integrating Meta-Information Management and Reflection. In *Middleware. WORDS* 2000.

[155] Scowen R. S. Extended BFN: A generic base standard. ISO/IEC 14977, 1996. http://www.cl.cam.ac.uk/~mgk25/iso-ebnf.html.

[156] V. Issarny, C. Bidan, and T. Saridakis. Achieving Middleware Customization in a Conguration-Based Development Environment : Experience with the Aster Prototype. *In Proceedings of the 4th International Conference on Configurable Distributed Systems, p. 207-214*, Annapolis, Maryland, USA, May 1998.

Apêndice A

A.1 Linguagem de Metaprogramação

Para apresentar a estrutura da linguagem de metaprogramação de aplicações definida para o Cosmos, este apêndice descreve em notação EBNF [155] os elementos do modelo de metacomponentes introduzido na seção 4.1.4. A estrutura da linguagem de metaprogramação contempla os conceitos definidos no protótipo original [76] [77]; adicionalmente, a linguagem é estendida com novas construções com o objetivo de dar suporte à especificação de requisitos de QoS. A linguagem é apresentada através de regras de produção que descrevem as sintaxes para declarações e diretivas de configuração dos elementos passíveis de configuração. Antes porém, são discutidos alguns conceitos fundamentais que serão usados nas definições das regras.

A.1.1 Os Elementos Básicos

Esta subseção descreve as definições dos elementos básicos da linguagem, os quais são apresentados nas regras da Fig. A.1 de acordo com a linguagem descrita em [76] [77].

```
<attributes>   ::= [<attribute>]+
<properties>   ::= [<property>]+
<constraints>  ::= [<constraint>]+
<attribute>    ::= attribute name=<name> value=<name>
<property>     ::= property name=<name> value=<lstname>
                          default=<name> order=<order>
<constraint>   ::= constraint name=<name> value=<lstname>
<lstname>      ::= <name>[,<name>]*
<name>         ::= [a-z0-9]+
<number>       ::= [0-9]+.[0-9]{1-2}
```

Fig. A.1**:** Regras que definem os elementos básicos da linguagem

As regras apresentadas na Fig. A.1 descrevem os seguintes símbolos definidos como *tags* em [76], [77]:

- *attributes*: símbolo que indica a definição de uma lista de atributos.

- *properties*: símbolo que indica a definição de uma lista de propriedades.
- *constraints*: símbolo que indica a definição de uma lista de restrições.
- *attribute*: especifica um atributo
    - *name*: nome do atributo.
    - *value*: valor associado ao atributo.
- *property*: especifica uma propriedade
    - *name*: nome da propriedade.
    - *value*: valor associado à propriedade, podendo ser uma lista.
    - *default*: valor padrão da propriedade.
    - *order*: ordem a ser tentada na seleção dos possíveis valores de uma propriedade. Seu valor poderá ser: *asc* e *desc*, denotando respectivamente o conceito de percurso na ordem ascendente e descendente, respectivamente.
- *constraint*: especifica uma restrição que está associada à uma determinada propriedade da aplicação; ou seja, no momento em que se define os componentes integrantes da aplicação, pode-se aplicar restrições às propriedades desses componentes.
    - *name*: nome da propriedade que esta sendo restringida.
    - *value*: valor a ser restringido, esse podendo ser uma lista de valores;
- *name*: especificação do padrão de nomes utilizado;
- *number*: especificação de números;

A.1.2 Especificação da Aplicação

A definição de um componente denominado *application* é utilizada para descrição de aplicações. De acordo com o modelo definido no Cosmos, uma aplicação é construída através de um processo de composição. Para isto, uma especificação deve indicar os componentes dos quais a aplicação depende e as conexões envolvidas entre as respectivas portas. O componente *application* é o elemento chave no processo de configuração, pois, a partir dele pode-se obter os demais elementos. A Fig. A.2 apresenta a sintaxe associada à descrição de uma aplicação.

```
<application>   ::= application name=<name> <depends> <connections>
<depends>       ::= [<component>]*
<component>     ::= component name=<name> description=<name>
                        location=<name> Instance = <name>
                        [<properties>]+ [<attributes>]+
                        [<ports>]* [<constraints>]*
<connections>   ::= [<connect>]+
<connect>       ::= connect from=<lstport> to=<lstport> <lstqos>
<lstport>       ::= <instanceName>:<port>[,<instanceName>:<port>]*
<lstqos>        ::= <qos>[,<qos>]*
<componentName> ::=  <name>
```

Fig. A.2: Regras para construção de aplicações

As regras apresentadas na Fig. A.2 descrevem os seguintes símbolos:

- *application*: símbolo inicial da configuração da aplicação;
    - *name*: especifica o nome da aplicação;
    - *properties*: tratamento da lista de propriedades;
    - *depends*: descreve a lista de componentes que faz parte da aplicação;
    - *component*: indica um componente na lista de dependência.
    - *connections*: descreve a lista de conexões de componentes da aplicação;
- *connection*: define ligações entre portas;
- *from*: descreve a lista de portas origem dos fluxos;
- *to*: descreve a lista de portas destino do fluxo;
- *instanceName*: nome da instância que contém a porta referenciada na conexão;
- *lstqos*: descreve a lista de elementos qos que tratam a questão de qualidade de serviço;
- *qos*: mostra como é tratada a questão de qualidade de serviço na conexão. A descrição deste símbolo pode ser melhor entendida através das ilustrações do exemplo apresentado na Seção 5.3.

Na descrição de uma aplicação, o atributo *name* define o nome da aplicação a ser executada no *middleware*. A especificação de uma aplicação contém duas partes fundamentais: dependências e conexões.

Na descrição das dependências, os componentes necessários à aplicação têm seus nomes e instâncias indicadas, bem como suas respectivas localizações, podendo ter restrições e substituição de valores de propriedades associadas.

Na parte de conexões encontra-se a descrição da forma como as portas se conectam. Na descrição de conexão indicam-se os atributos *to* e *from*. O atributo *from* informa que a porta associada é uma porta de saída – ou seja, porta que coloca dados na conexão. O atributo *to* indica que a porta associada é uma porta de entrada, ou seja, que recebe informações. É também na conexão onde são definidos os elementos para tratar as questões de qualidade de serviço, conforme ilustrado adiante.

A.1.3 Especificação de Componente

Aplicações são compostas de componentes que são efetivamente os elementos responsáveis pelo funcionamento da aplicação. Os componentes têm como atributos seu nome, descrição e localização, consistindo a especificação de três partes: propriedades, restrições e portas.

As propriedades são armazenadas como pares (chave, valor), podendo o valor corresponder a uma lista de possibilidades. As portas são elementos que descrevem os componentes responsáveis por transmitir e receber fluxo de dados. A sintaxe para descrição do elemento componente é apresentada na Fig. A.3 a seguir:

```
<component>   ::= component name=<name> description=<name>
                  location=<namelocation> [<properties>]+ [<attributes>]+
                  [<ports>]* [<constraints>]*
```

Fig. A.3: Especificação do componente

A regra apresentada na Fig. A.3 descreve os seguintes símbolos:

- *component* - símbolo que indica o início da seção destinada a descrever o componente
    - *name* - indica o nome do componente. Cada componente deve ter um nome único dentro do sistema.
    - *description* - descreve o funcionamento do componente.
    - *location* – indica a localização do componente.
    - *properties* - seção usada para descrever as propriedades do componente.

- *ports* – seção usada para descrever a lista de portas; a sintaxe para descrição de uma porta é descritoa adiante, na Seção A.1.4
- *constraints* - descreve as restrições relacionadas com as propriedades associadas a um fluxo gerado pelo componente.

A.1.4 Especificação de Porta

No *middleware*, os componentes recursos virtuais podem ter portas. Eventualmente podem existir componentes que não precisem de portas para seu funcionamento; no entanto, para que haja fluxos de comunicação multimídia entre os componentes devem-se agregar portas, as quais serão vinculadas às conexões na aplicação.

Uma porta tem associado um nome, um tipo e uma lista de propriedades. Sua descrição é mostrada a seguir, na Fig. A.4:

```
<port>   ::= port name=<name>
             type=<porttype>
             <properties>
```

Fig. A.4: Especificação da porta.

A regra apresentada na Fig. A.4 descreve os seguintes símbolos:

- *port*: símbolo que indica o início da configuração da porta
  - *name*: nome associado à porta.
  - *porttype*: uma indicação do tipo da porta (*inputPort* ou *outputPort*).
  - *properties*: lista de propriedades ligadas à porta.

A.1.5 Especificação de QoS

O modelo de gerenciamento e controle de QoS proposto para o Cosmos [83] [84], cujos aspectos principais são detalhados no Capítulo 4, faz menção ao processo de configuração de parâmetros de QoS associados às conexões. Para suportar adaptação dinâmica, este modelo define componentes arquiteturais ativos para verificar se os parâmetros de QoS do sistema estão de acordo com os requisitos. Face à grande flutuação causada por fatores inerentes à aplicação, plataforma e

infra-estrutura de comunicação, o modelo utiliza o conceito de intervalos de QoS, no qual a definição dos requisitos é feita especificando faixas de valores. Caso seja detectada durante a execução alguma situação de não cumprimento de QoS negociada, um processo de adaptação é iniciado observando os requisitos estabelecidos na descrição da aplicação. De acordo com o modelo definido, o controle da QoS se concentra no monitoramento das conexões observando os parâmetros de QoS indicados na especificação; o controle consiste em verificar se os parâmetros estão fora das faixas especificadas. Para tal, é necessário indicar quais os parâmetros que precisam ser monitorados, definir as regiões aceitáveis de QoS, estabelecer uma região padrão, bem como a taxa de observação e o tempo (freqüência) em que deve se tentar mudar de região.

A sintaxe básica para descrever os requisitos de QoS é apresentada na Fig. A.5, a seguir:

```
<qos>           ::= qos parameter=<name> <defaultregion> <frequency> <testtime> <regions>
<defaultregion> ::= defaultregion=<name>
<frequency>     ::= frequency=<number>
<testtime>      ::= testtime=<number>
<regions>       ::= [<region>]+
<region>        ::= region name=<name> <range> <default_Property>[<constraint>]*
<range>         ::= range min=<number> max=<number>
<default_Property>::= name = <nome-property> values = <number>
```

Fig. A.5: Especificação de QoS

As regras apresentadas na Fig. A.5 descrevem os seguintes símbolos:

- *qos*: símbolo que indica o início da configuração de um parâmetro de qualidade de serviço
  - *parameter*: indica qual será o parâmetro de QoS a ser monitorado.
  - *defaultregion*: especifica a região default.
  - *frequency*: diz qual será a freqüência de observação.
  - *testtime*: indica o período a ser tentado mudar de região.
  - *regions*: denota as regiões definidas;
- *region*: descreve uma região
  - *name*: define o nome da região.
  - *range*: define os valores limites da região
    - *min* - valor que indica o limite inferior da região; se omitido denota que a região contempla todos os valores abaixo do limite superior.

- *max* - valor que indica o limite superior da região; se omitido, denota que a região contempla todos os valores acima do limite inferior.

- *default_property*: define um valor *default* para a propriedade indicada na região.

A.1.6 Descrição Completa da Linguagem

A Fig. A.6 apresenta uma visão geral da linguagem de descrição de aplicações envolvendo os elementos e definições discutidos anteriormente. A partir desta linguagem, novos *parsers* podem ser desenvolvidos, desde que sejam mantidos os conceitos incorporados na linguagem, bem como seus respectivos mapeamentos para a hierarquia de metadados.

```
<application>   ::= application name=<name> <depends> <connections>
<depends>       ::= [<component>]*
<component>     ::= component name=<name> description=<name>
                    location=<name> Instance = <name>
                    [<properties>]+ [<attributes>]+
                    [<constraints>]* [<ports>]*
<ports>         ::= [<port>]+
<port>          ::= port name=<name> type = <porttype> <properties>
<porttype>      ::= outputport | inputport
<connections>   ::= [<connect>]+
<connect>       ::= connect from=<lstport> to=<lstport> <lstqos>
<lstport>       ::= <instanceName>:<port>[,<instanceName>:<port>]*
<lstqos>        ::= <qos>[,<qos>]*
<componentName> ::=<name>
<qos>           ::= qos parameter=<name> <defaultregion> <frequency> <testtime> <regions>
<defaultregion> ::= defaultregion=<name>
<frequency>     ::= frequency=<number>
<testtime>      ::= testtime=<number>
<regions>       ::= [<region>]+
<region>        ::= region name=<name> ,<range> <default_Property>[<constraint>]*
<range>         ::= range min=<number> max=<number>
<attributes>    ::= [<attribute>]+
<properties>    ::= [<property>]+
<constraints>   ::= [<constraint>]+
<default_Property>::= name = <nome-property> values = <number>
<attribute>     ::= attribute name=<name> value=<lstname> default=<name>
                    order=<order>
<property>      ::= property name=<name> value=<lstname> default=<name>
                    order=<order>
<constraint>    ::= constraint name=<name> value=<lstname>
<lstname>       ::= <name>[, <name>]*
<name>          ::= [a-z0-9_]+
<number>        ::= [0-9]+.[0-9]{1-2}
<order>         ::= asc | desc
```

Fig. A.6**:** Visão Geral da linguagem de metaprogramação

Apêndice B

B.1 XML-Schemas para a Linguagem de Especificação do *middleware* AdapTV

Para o AdapTV, uma aplicação é descrita e tratada explorando o conceito de composição de acordo com o modelo de especificação definido no Cosmos. A seguir, este apêndice apresenta o modelo de especificação que foi definido para o AdapTV com base na linguagem XML. Os *XML Schemas* apresentados adiante correspondem aos elementos descritos no modelo UML ilustrados na Fig. 5.4 .

B.1.1 Bloco de Especificação de um Componente

Como elemento base do *XML-Schema,* a Fig. B.1 descreve o bloco *component*, definindo a *tag* raiz *component*.

```
...
<xs:element name="Component">
    <xs:complexType>
        <xs:sequence>
                <xs:element ref="attributes"/>
                <xs:element ref="depends"/>
                <xs:element ref="properties"/>
                <xs:element ref="ports"/>
        </xs:sequence>
    </xs:complexType>
 <xs:attribute ref="name"/>
 <xs:attribute ref="description"/>
 </xs:element>
```

Fig. B.1: Bloco *component*

Como pode ser visto na Fig. B.1 um componente pode ter os seguintes elementos:

- ***ATTRIBUTES:***: enumera os atributos do componente descrito.
- ***DEPENDS***: identifica os componentes dos quais o componente descrito depende.

- ***PROPERTIES***: identifica os atributos não funcionais (ou propriedades).
- ***PORTS***: identifica as portas de entrada e saída do componente descrito;

A descrição dos atributos de um componente tem uma estrutura similar à descrição de propriedades. Uma propriedade é descrita através de um conjunto de valores para os quais aquela propriedade poderá ser instanciada. Para um conjunto de possíveis valores v_1, v_2, v_3, ... , v_n, as formas possíveis para especificá-lo são:

- v_1 .. v_n : conjunto de valores desde v_1 até v_n.
- v_1, v_2, v_n : conjunto contendo apenas v_1, v_2 e v_n
- v_1 .. v_2, v_m .. v_n : conjunto contendo de v_1 a v_2, e v_m a v_n, sendo $v_m < v_n$

Desta forma, estabelece-se um conjunto finito de valores para os quais aquela propriedade poderá ser instanciada. A ordem de preferência (usada por exemplo, para expressar o nível de QoS que interessa à aplicação) dos elementos deste conjunto de valores é definida de acordo com a *tag order*, que pode assumir os valores a*sc* ou *desc*, indicando ao *Configurator*, que no processo de escolha, dê prioridade aos elementos do início ou fim deste conjunto, respectivamente.

B.1.2 Bloco de Especificação de Portas

O bloco *ports*, descrito na Fig. B.2 define a sintaxe para especificação de portas.

```
<xs:element ref="port">
    <xs:complexType>
        <xs:element ref="property"
                    maxOccurs ="unbounded"/>
    </xs:complexType>
    <xs:attribute ref="name" use="required"/>
    <xs:attribute ref="type" use="required"/>
</xs:element>
<xs:element name="ports">
    <xs:complexType>
        <xs:element ref="port"
                    maxOccurs ="unbounded"/>
    </xs:complexType>
</xs:element>
```

Fig. B.2: Bloco *ports*

Cada porta é identificada por uma cadeia de caracteres (*name*), tendo um conjunto de propriedades. Propriedades identificam características da porta associada, como por exemplo, tamanho do buffer e taxa de transmissão.

B.1.3 Bloco de Especificação de Conexões

O bloco *connections* (Fig. B.3) define a sintaxe para especificar conexões onde estão envolvidas as indicações das portas relacionadas, os requisitos de QoS e as estratégias de adaptação. A descrição do bloco de conexão é apresentado em duas etapas. Inicialmente, a Fig. B.3 apresenta a estrutura básica de uma conexão, onde devem ser identificadas a origem (atributo *from*) e o destino (atributo *to*) da conexão; os nomes dos componentes indicados e das portas devem ser separados por vírgulas.

```
<xs:simpleType name="location">
<xs:restriction base="xs:string">
    <xs:pattern value=
            "([a-z]|[0-9])*:[a-z]|[0-9])*"/>
</xs:restriction>
</xs:simpleType>
<xs:element name="from" type="location"/>
<xs:element name="to" type="location"/>
<xs:element name="connect">
  <xs:complexType>
    <xs:attribute ref="from" use="required"/>
<xs:attribute ref="to" use="required"/>
  </xs:complexType>
</xs:element
<xs:element name="connections">
  <complexType>
    <xs:element ref="connect"
      <xs:element name="QoS">
                ●●●
                maxOccurs="unbounded"/>
  </complexType>
</xs:element>
```

Fig. B.3: Bloco básico de conexão

A Fig. B.4 consiste basicamente em uma extensão do bloco de conexão introduzido na figura anterior onde são apresentados os elementos definidos para especificação de requisitos de QoS e adaptação dinâmica em conexões virtuais. A separação do bloco em duas figuras tem como objetivo simplificar a apresentação e ao mesmo tempo destacar os mecanismos de especificação de QoS.

Os principais elementos introduzidos no esquema da relacionados com gerenciamento de QoS em conexões virtuais, são os seguintes:

- ***PROPERTY***: freqüentemente usado para descrição de requisitos não funcionais, como parâmetros de QoS associados a *jitter* e retardo da rede.
- ***parameter***: atributo que identifica um parâmetro de QoS a ser monitorado.

```
...
<xs:element name="QoS">
    <xs:attribute ref="parameter" use="required"/>
        <xs:complexType>
            <xs:element name="DEFAULTREGION" type ="xs:string"
                                        use="required" />
            <xs:element name="FREQUENCY" type= "xs:decimal"
                                        use="required" />
            <xs:element name="TESTTIME" type ="xs:decimal"
                                        use="required" />
            <xs:element name="REGIONS">
                <xs:complexType>
                    <xs:element name="REGION" maxOccurs="unbound">
                            <xs: attribute name="name" type="xs:string"
                        use="required/>
                    <xs:complexType>
                        <xs:element name="PROPERTY" maxOcurs ="unbound"/>
                        <xs:element name="RANGE">
                                    <xs:attribute name="max" type ="xs:string" />
                                    <xs:attribute name="min" type ="xs:string" />
                        </xs:element>
                        <xs:element name="PROPERTY_DEFAULT">
                                    <xs:attribute name="name" type ="xs:string" />
                                    <xs:attribute name="value" type ="xs:string" />
                        </xs:element>
        ...
```

Fig. B.4: XML-Schema para descrição de QoS no bloco de conexão

- ***DEFAULTREGION***: atributo requerido para especificar a região de QoS para a operação inicial, sendo definida como padrão para o recurso monitorado.
- ***FREQUENCY***: Atributo usado para especificar a freqüência de observação da QoS.
- ***TESTTIME***: atributo utilizado para especificar, em milisegundos, a periodicidade em que o componente *Configurator* deve tentar mudar da região atual de QoS para outra qualitativamente melhor, ou para à região padrão (definida através do atributo *DefaultRegion*).
- ***REGIONS***: denotam as regiões de QoS para efeito de operação, as quais são definidas por elementos que especificam seus limites superior e inferior.
- ***RANGE***: determina uma faixa de QoS associada a uma região.
- ***PROPERTY_DEFAULT***: determina um valor *default* de propriedade associado à região.

Apêndice C

C.1 Descrição das Classes de Implementação do *middleware* AdapTV

Neste apêndice são discutidas as classes apresentadas na Fig. 5.9 relacionadas a processamento de metadados e aos modelos de QoS e de Interconexão de Componentes.

C.1.1 Processamento de metadados

A classe *ApplicationMetadata* é responsável por manter e disponibilizar as informações de configuração dos componentes e conexões da aplicação. A Fig. C.1 descreve as assinaturas das operações definidas nesta classe, cujas descrições são brevemente discutidas a seguir:

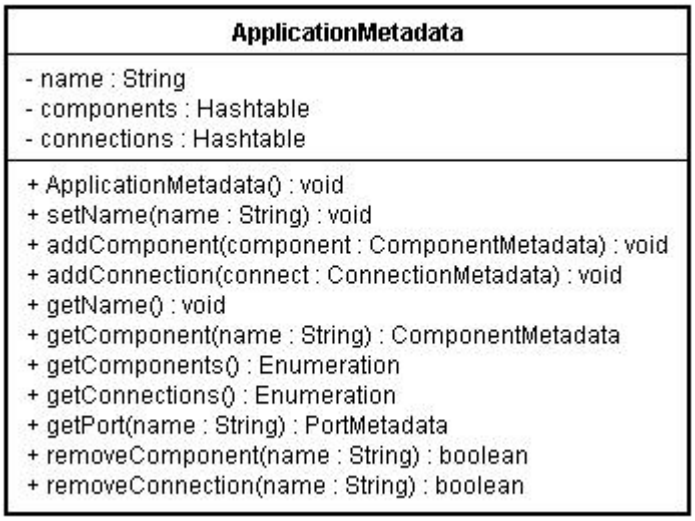


Fig. C.1: Classe *ApplicationMetadata*

- *ApplicationMetadata*: inicia as listas associadas a componentes e conexões;
- *setName*: define o nome do elemento que está sendo processado, colocando o identificador definido no atributo *name*;

- *addComponent*: adiciona um componente à aplicação. Esse componente é incluído no atributo *components*;
- *addConnection*: adiciona uma conexão à aplicação. Essa conexão é incluída no atributo *connections*;
- *getName*: retorna o nome da aplicação que está no atributo *name*;
- *getComponent*: dado o nome do componente, verifica se ela está presente, retornando o *componentMetadata* caso este tenha sido incluído, ou nulo se não for encontrado;
- *getComponents*: retorna um *iterator* do tipo *Enumeration* que acessa a lista de componentes da aplicação;
- *getConnections:* retorna um *iterator* do tipo *Enumeration* que acessa a lista de conexões da aplicação;
- *getPort:* dado o nome da porta, é feita uma pesquisa nos componentes, retornando o componente *portMetadata*; caso não tenha sido descrita esta porta, é retornado *null*.
- *removeComponent:* a partir do nome do componente é feita a tentativa para remover o componente da aplicação, caso seja possível, é retornado o valor *true*, caso contrario, *false*. No momento de remover o componente, deve-se verificar se existe algum componente que dependa dele;
- *removeConnection:* dado o nome da conexão, é realizada a remoção da mesma.

A classe *ComponentMetadata* é responsável por guardar e disponibilizar as características dos componentes envolvendo a lista de portas, propriedades e localização, entre outros. A Fig. C.2 mostra os métodos definidos para esta classe:

- *ComponentMetadata*: cria as listas responsáveis por armazenar os atributos, propriedades, restrições, portas e componentes do quais ele depende;
- *setName*: faz a associação do nome do componente ao atributo *name*;
- *setDescription*: faz a associação da descrição do componente ao atributo *description*;
- *setInstance*: faz a associação do nome da instância do componente, ao atributo *instance*;
- *addLocation*: adiciona à lista de localizações um possível local onde o componente pode ser encontrado;
- *addProperty*: adiciona uma nova propriedade ao componente, incluído-a na lista *property*;

- *addPort*: adiciona uma nova porta ao componente, incluindo-a na lista *ports*;
- *getName*: retorna o nome do componente;

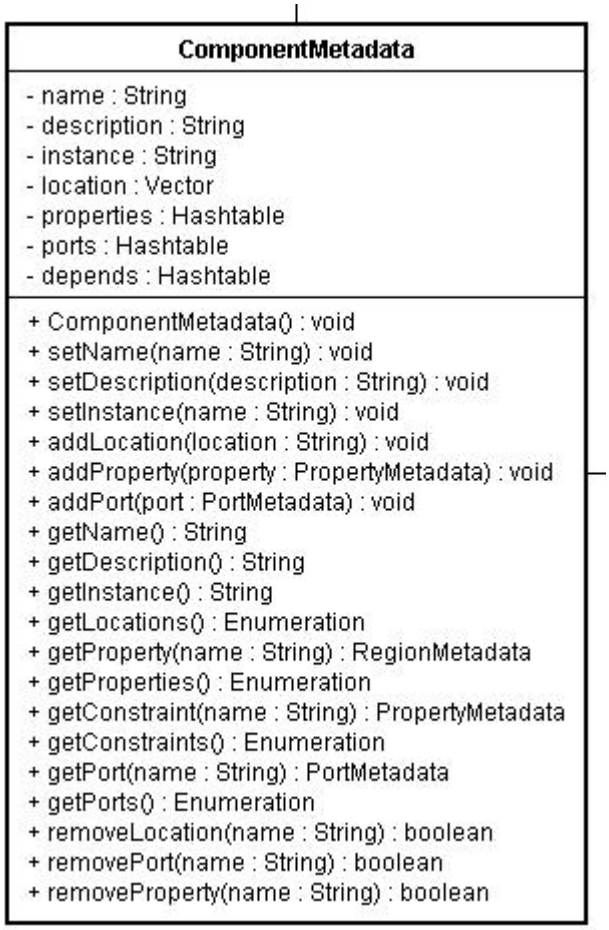


Fig. C.2 : Classe *ComponentMetadata*

- *getDescription*: retorna a descrição do componente;
- *getInstance*: retorna o nome definido e guardado no atributo *instance*; esse nome é utilizado para identificar a instância do componente a ser realizada;
- *getLocation*: retorna uma enumeração que pode ser utilizada para acesso à lista de localizações do componente descrito;
- *getProperty*: retorna a propriedade solicitada, caso não exista, esse valor será nulo;
- *getProperties*: retorna uma enumeração que dá acesso aos elementos da lista de propriedades do componente;
- *getPort*: dado o nome de uma porta, é feita uma pesquisa na lista e retorna a porta; caso a porta não exista, é retornado o valor nulo.

- *getPorts*: retorna uma enumeração que dá acesso aos elementos da lista de portas que estão associadas ao componente;
- *removeLocation*: dado uma localização, é feita a remoção da mesma;
- *removeProperty:* dado o nome da propriedade, é feita a remoção;
- *removePort:* dado o nome da porta, essa é excluída do componente;

A classe *ConnectionMetadata* é responsável por guardar os metadados de uma conexão. Nela existem três listas que guardam as portas de origem, de destino e a QoS associada. A Fig. C.3 mostra a assinatura com operações desta classe e, em seguida, o texto apresenta uma descrição de cada operação:

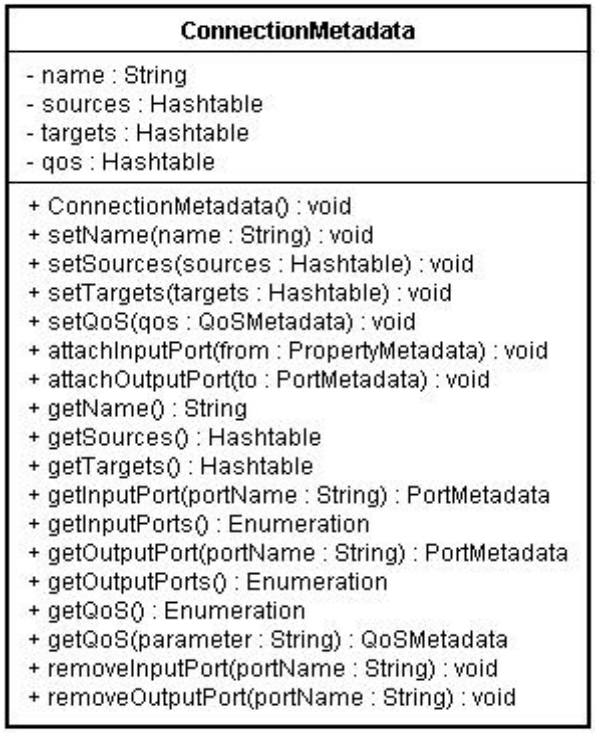


Fig. C.3: Classe *ConnectionMetaData*

- *ConnectionMetadata*: faz a iniciação das listas *sources* e *targets*;
- *setName*: associa um nome à conexão;
- *setSources*: faz a associação de uma lista já existente de portas de origem;
- *setTargets*: faz a associação de uma lista já existente de portas de destino;
- *setQoS*: adiciona um parâmetro de QoS à lista de QoS;

- *attachInputPort*: adiciona uma porta à lista de portas de origem;
- *attachOutputPort*: adiciona uma porta à lista de portas de destino;
- *getName*: retorna o nome que está associado à conexão;
- *getSources*: retorna a lista de portas de origem da conexão;
- *getTargets*: retorna a lista de portas de destino da conexão;
- *getInputPort*: dado o nome de uma porta, é feita uma pesquisa e retornada a porta solicitada, caso não exista, é retornado o valor nulo;
- *getInputPorts*: retorna uma enumeração para acesso à lista de portas de origem;
- *getOutputPort*: dado o nome de uma porta é feita uma pesquisa e retornada a porta solicitada, caso não exista, é retornado o valor nulo;
- *getOutputPorts*: retorna uma enumeração para acesso à lista de portas de destino.
- *getQoS*: retorna uma enumeração para acesso à lista de QoS da conexão; caso a chamada especifique um parâmetro, a operação retorna o elemento de QoS correspondente;
- *removeInputPort*: remove a porta especificada da lista de portas de origem;
- *removeOutputPort*: remove a porta especificada da lista de portas de destino;

A classe *PortMetadata* é responsável por armazenar os nomes, as propriedades e as restrições associadas às portas. Essas propriedades e restrições são armazenadas em listas. A seguir, a Fig. C.4 mostra as assinaturas das operações definidas na classe:

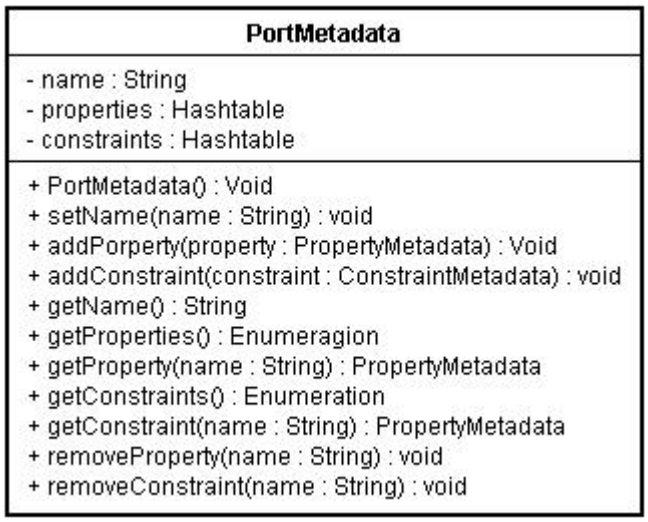


Fig. C.4: Classe *PortMetadata*

- *PortMetadata*: cria as listas que armazenarão as propriedades e restrições da porta;
- *setName*: associa o nome do componente ao atributo *name*;
- *addProperty*: adiciona uma nova propriedade à lista *properties* da porta;
- *addConstraint*: adiciona uma nova restrição à lista *constraints* da porta;
- *getName*: retorna o nome da porta;
- *getProperties*: retorna uma enumeração que dá acesso às propriedades contidas na lista de propriedades;
- *getProperty*: dado o nome de uma propriedade, retorna os metadados associados à propriedade;
- *getConstraints*: retorna uma enumeração que dá acesso às restrições contidas na lista de restrições.
- *removeProperty*: dado o nome da propriedade, é feita a sua remoção da lista de propriedades;
- *removeConstraint:* dado o nome da restrição, é feita a sua remoção da lista de restrições;

A classe *PropertyMetadata* define uma estrutura a ser utilizada para guardar propriedades e restrições. A Fig. C.5 mostra as assinaturas dos métodos da classe e, em seguida, uma breve descrição dos métodos:

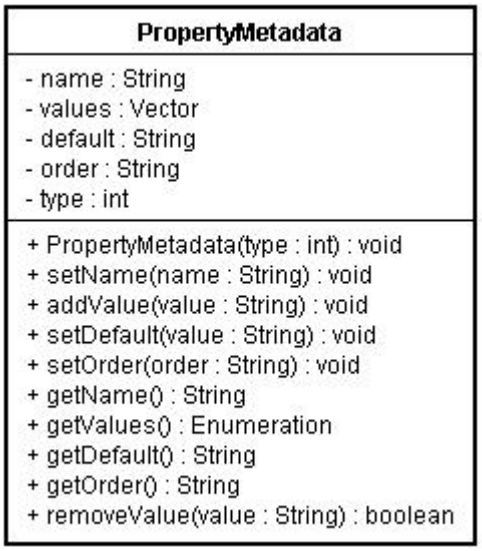


Fig. C.5: Classe *PropertyMetadata*

- *PropertyMetadata*: cria a lista que contém os valores associados ao metadado;
- *setName*: faz a associação do nome do metadado ao atributo *name*;
- *addValue*: adiciona um possível valor à lista de valores;
- *setDefault*: faz a associação do valor *default*;
- *setOrder*: diz que ordem deve ser seguida na escolha dos valores;
- *getName*: retorna o nome do metadado;
- *getValues*: retorna uma enumeração para acesso à lista de valores;
- *getDefault*: retorna o valor *default*;
- *getOrder*: retorna a ordem de pesquisa dos valores associados ao metadado;
- *removeValue*: remove o valor indicado no parâmetro dos valores associados à propriedade.

A classe *QoSMetadata* é responsável por manter os elementos que descrevem como o gerenciador de QoS deve atuar. Estes elementos no protótipo estão associados à conexão. A Fig. C.6. mostra as assinaturas das operações definidas para esta classe:

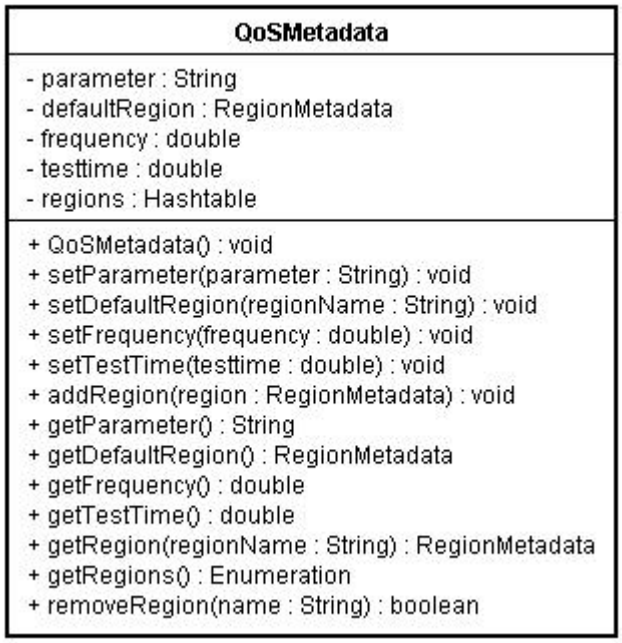


Fig. C.6: Classe *QoSMetadata*

- *QoSMetadata*: cria a lista a lista de restrições;
- *setParameter*: faz a associação do parâmetro ao QoS;

- *setDefaultRegion*: dado o nome da região, essa é associada como *default*;
- *setFrequency*: diz qual é a freqüência de amostragens;
- *setTestTime:* diz qual é o tempo para se fazer uma amostragem;
- *addRegion:* adiciona uma região ao QoS;
- *getParameter:* retorna o parâmetro de QoS;
- *getDefaultRegion:* retorna a região *default*;
- *getFrequency:* retorna a freqüência de amostragem;
- *getTestTime:* retorna o tempo (período) em que devem ser feitas tentativas de mudança de regiões;
- *getRegions:* retorna uma enumeração para acesso à lista de regiões;
- *getRegion:* retorna os metadados associados à região indicada no parâmetro;
- *removeRegion:* dado o nome de uma região, é realizado a sua remoção;

A classe *RegionMetadata* é responsável por manter as descrições que definem regiões de operação, correspondendo aos valores de limite inferior e limite superior de QoS. Na implementação, estes dados são gerenciados pela classe definida na Fig. C.7 cujas operações são descritas a seguir:

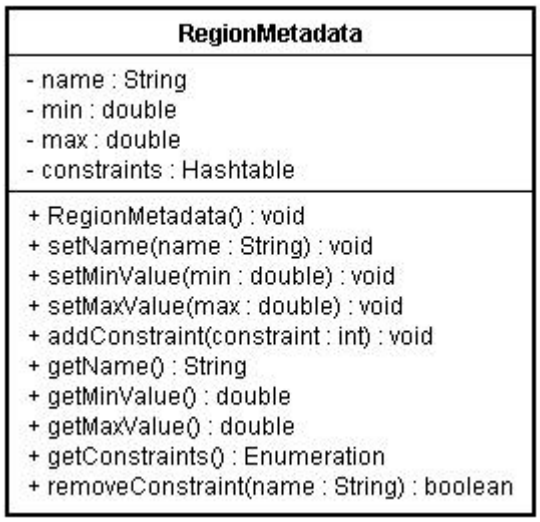


Fig. C.7: Classe *RegionMetadata*

- *RegionMetadata*: cria a lista que contém as restrições;
- *setName*: faz a associação do nome do metadado à região;

- *setMinValue*: especifica o valor mínimo da região;
- *setMaxValue*: especifica o valor máximo da região;
- *addConstraint*: adiciona uma restrição à lista;
- *getName*: retorna o nome da região;
- *getMinValue*: retorna o valor mínimo da região;
- *getMaxValue*: retorna o valor máximo da região;
- *getConstraints*: retorna uma enumeração para acesso à lista de restrições;
- *removeConstraint*: dado o nome de uma restrição é realizado a sua remoção.

De acordo com a discussão apresentada na Seção 5.2.3.1, a implementação de um *parser* deverá prover as interfaces *IParserCreator,* o construtor real, e *IParser*, o componente instanciado pelo construtor.

A classe *XMLParserCreator* é responsável por criar o *parser* específico que foi definido para XML. Seu papel na implementação é o de uma fábrica concreta para o *parser* e sua assinatura é apresentada na Fig. C.8:

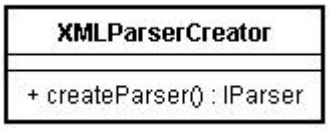


Fig. C.8: Fábrica de *parser* XML

- *createParser*: cria um objeto do tipo XMLParser, o qual implementa a interface IParser, retornando-o.

A assinatura do *parser* concreto definido para implementação do protótipo, o XMLParser, está descrita na Fig. C.9. Seu papel é processar uma especificação baseada na sintaxe XML e gerar os metadados para a aplicação. Essa classe implementa a interface *IParser* que é conhecida pelo *Configurador.*

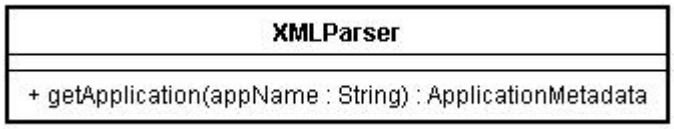


Fig. C.9: Classe *XMLParser*

- *getApplication*: implementação do método da interface IParser que é responsável por processar os arquivos de descrição e gerar os metadados para ApplicationMetadata, descrevendo assim todos os elementos envolvidos.

O processamento dos arquivos que descrevem uma aplicação é feito pela classe *XMLApplicationReader*. Ele é responsável por criar e retornar o metadado da aplicação. Sua assinatura é descrita na Fig. C.10, consistindo nos seguintes métodos:

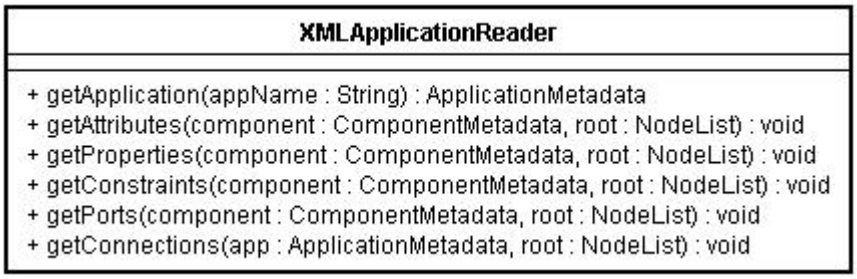


Fig. C.10: Classe *XMLApplicationReader*

- *getApplication*: retorna o componente e a estrutura com os metadados da aplicação;
- *getAttributes*: método que faz o processamento de um elemento associado à *tag* raiz de uma especificação XML obtendo os atributos de um componente. Esses atributos são encontrados no arquivo de configuração da aplicação;
- *getProperties*: método que faz o processamento de um elemento associado à *tag* raiz de uma especificação XML obtendo as propriedades de um componente. Esses elementos são encontrados no arquivo de configuração da aplicação;
- *getConstraints*: método responsável por processar as restrições pertencentes a um componente;
- *getPorts*: método que faz o processamento das portas dos componentes;
- *getConnections*: método que monta as conexões entre os componentes e faz a configuração dos parâmetros para o QoS.

Como o *XMLApplicationReader*, o *XMLComponentReader* faz a leitura dos arquivos que descrevem os componentes da aplicação. A descrição da classe *XMLComponentReader* é apresentada na Fig. C.11, consistindo nos seguintes métodos:

- *getComponent*: retorna a estrutura com os metadados do componente;
- *getAttributes*: faz o processamento dos atributos do componente que estão no arquivo de descrição do componente;
- *getProperties*: as propriedades são processadas e incluídas no componente;
- *getPorts*: adquire as portas dos componentes;

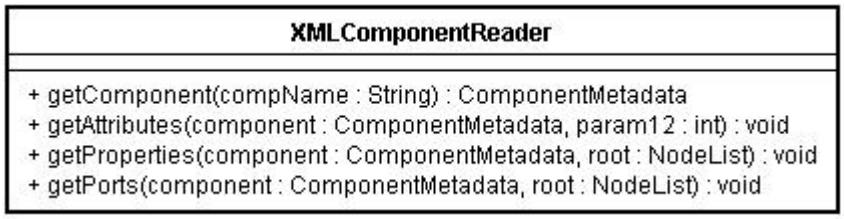


Fig. C.11: Classe *XMLComponentReader*

C.1.2 Classes do modelo de QoS

A implementação do modelo de QoS utiliza intensamente a hierarquia de classes definida para processamento de metadados discutida na seção anterior. As classes definidas para implementação do modelo correspondem aos componentes do modelo descrito em [83] [84]. Um diagrama UML com as classes e operações das interfaces definidas no modelo de QoS é apresentado na Fig. C.12.

A classe QoS *Manager* implementa as interfaces *IManagerControl* e *IManager* descritas na Seção 4.2.3. A interface *IMonitorControl* é implementada pela classe *QoSMonitor*; as operações desta interface encontram-se descritas na Fig. 4.25. A interface *IStatusControl* foi definida no escopo do modelo de componentes Cosmos, ou seja, as operações desta interface devem ser implementadas em todos os componentes de um sistema baseado no *framework* Cosmos. A sua descrição é apresentada na página 64. A interface *IMonitored* deve ser implementada apenas em componentes passíveis de realização de monitoramento de seu estado. A descrição das operações associadas a esta interface encontra-se descrita na Fig. 4.24.

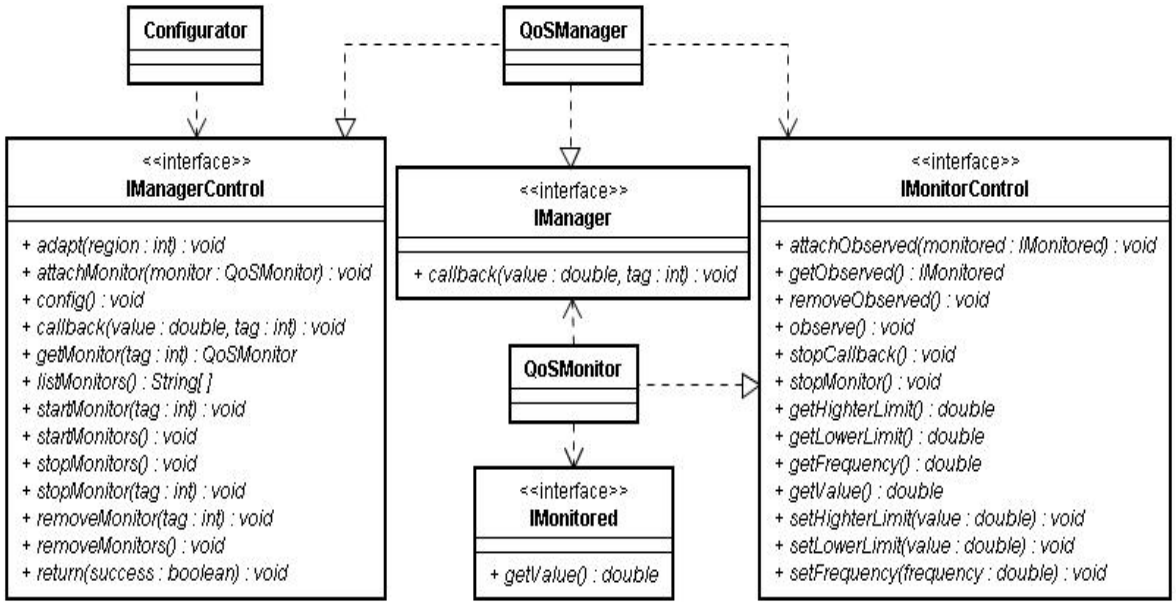


Fig. C.12: Diagrama de classes do modelo de QoS com as operações das interfaces associadas

C.1.3 Classes do Modelo de Interconexão

As principais classes que compõem a arquitetura de implementação do modelo de interconexão foi discutida na Seção 4.2.4.