Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Programa de Pós-Graduação em Sistemas e Computação

Dissertação de Mestrado

Um Modelo de Interconexão de Componentes para Ambientes Multimídia Distribuídos

Carlos Eduardo da Silva

Natal - RN
Fevereiro de 2007

Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Programa de Pós-Graduação em Sistemas e Computação

Um Modelo de Interconexão de Componentes para Ambientes Multimídia Distribuídos

Dissertação submetida ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte como requisito para a obtenção do grau de Mestre em Sistemas e Computação (MSc.).

Carlos Eduardo da Silva

Prof. Dr. Glêdson Elias da Silveira
Orientador

Prof. Dr. Adilson Barboza Lopes
Co-orientador

Natal – RN
Fevereiro de 2007

Catalogação da Publicação na Fonte. UFRN / SISBI / Biblioteca Setorial Especializada do Centro de Ciências Exatas e da Terra – CCET.

Silva, Carlos Eduardo da.
Um modelo de interconexão de componentes para ambientes mulitmídia distribuídos / Carlos Eduardo da Silva. – Natal, 2007.
115 f. : il.

Orientador: Glêdson Elias da Silveira.
Co-orientador: Adilson Barboza Lopes.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Departamento de Informática e Matemática Aplicada. Programa de Pós-Graduação em Sistemas e Computação.

1. Middleware – Dissertação. 2. Componentes de software – Dissertação. 3. Sistema multimídia distribuídos – Dissertação. 4. Interconexão de componentes – Dissertação. I. Silveira, Gledson Elias da. II. Lopes, Adilson Barboza. III. Título.

RN/UF/BSE-CCET CDU 004.4’27

Um Modelo de Interconexão de Componentes para Ambientes Multimídia Distribuídos

Carlos Eduardo da Silva

’Esta dissertação de mestrado foi avaliada e considerada aprovada pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte.’

_____
Prof. Dr. Glêdson Elias da Silveira
Orientador

_____
Prof. Dr. Adilson Barboza Lopes
Co-Orientador

Banca Examinadora:

_____
Prof. Dr. Glêdson Elias da Silveira
Presidente

_____
Prof. Dr. Adilson Barboza Lopes

_____
Profa. Dra. Thais Vasconcelos Batista

_____
Profa. Dra. Maria da Graça Campos Pimentel

Agradecimentos

Agradeço a minha família, pela formação e por todo apoio recebido durante esta árdua jornada, que se iniciou há muito tempo atrás e que ainda tem muito pela frente.

Aos meus orientadores e amigos, Professores Gledson e Adilson, por terem acreditado em mim para a realização deste trabalho. Ambos me acompanham desde a graduação, e com certeza contribuíram para a minha formação como pessoa durante estes anos que trabalhamos juntos. Agradeço pelo apoio e principalmente pela confiança depositada em mim durante a realização deste trabalho.

Agradeço a Márjory, por estar comigo durante todos os momentos, me dando apoio e não me deixando desanimar. Por me aturar durante os momentos estressantes, e pelos momentos em que deixamos de estar juntos para que este trabalho se realizasse.

Agradeço a Frederico Borelli (Arrobinha) e Milano, por terem iniciado este trabalho e implementado a primeira versão do protótipo, e a todos os envolvidos neste projeto, Fred mudo, Fred Lopes, Pow, Gugu e MacGyver que contribuíram de alguma maneira no desenvolvimento de nosso projeto.

Aos membros da banca examinadora, Professoras Thaís Batista e Maria da Graça Pimentel, por terem aceitado o convite para a banca e pelas sugestões apresentadas. A Professora Thaís pelo esforço empreendido para que a defesa pudesse ser realizada em tão pouco tempo. A Professora Graça por ter enfrentado um dia inteiro de correria para a realização desta defesa.

Agradeço aos membros dos laboratórios que compõem o Departamento de Informática e Matemática Aplicada, pelas muitas horas de T.O. que me proporcionaram. Aos membros e agregados do laboratório NATALNET, onde tudo começou. Ao Prof. Ivan pelo apoio dado a este projeto, tanto ao Prof. Adilson, quanto a mim como coordenador do laboratório NATALNET. Agradeço também os membros do laboratório CariocaNet que foram crucial durante a fase final deste trabalho. Agradeço, também, aos meus amigos Ivanilson Ninho, Jorgiano e Fellipão pelos memoráveis almoços no MidWay e cafés no São Bras.

Agradeço também ao Professor Guido Lemos, que sempre esteve presente no desenvolvimento deste trabalho através dos apoios fornecido ao Prof. Adilson e a mim pelos projetos HITV e FlexTV.

RESUMO

Sistemas multimídia devem incorporar conceitos de *middleware* de forma a abstrair especificidades de hardware e sistemas operacionais. Aplicações nestes sistemas podem ser executadas em diferentes tipos de plataformas, e os componentes destes sistemas precisam interagir uns com os outros. Neste contexto, faz-se necessário a definição de mecanismos de comunicação específicos para a transmissão de fluxos de informação. Este trabalho apresenta um modelo para a interconexão de componentes em ambientes multimídia, e sua arquitetura de implementação. O modelo oferece mecanismos de comunicação específicos para a transmissão de fluxos de informação entre componentes de software atendendo aos requisitos do *framework* Cosmos de maneira a suportar a reconfiguração dinâmica de componentes.

Palavras-chaves: *middleware*, componentes de software, sistemas multimídia distribuídos, interconexão de componentes.

ABSTRACT

Multimedia systems must incorporate middleware concepts in order to abstract hardware and operational systems issues. Applications in those systems may be executed in different kinds of platforms, and their components need to communicate with each other. In this context, it is needed the definition of specific communication mechanisms for the transmission of information flow. This work presents a interconnection component model for distributed multimedia environments, and its implementation details. The model offers specific communication mechanisms for transmission of information flow between software components considering the Cosmos framework requirements in order to support component dynamic reconfiguration.

Key-Words: middleware, software component, multimedia systems, component interconnection.

Sumário

Lista de Figuras

1. Introdução

Devido à integração das tecnologias de telecomunicação com a computação e a multimídia, o computador pode ser caracterizado como um veículo de comunicação. Esta convergência digital tem ocorrido principalmente devido à popularidade da Internet, onde o caráter distribuído das aplicações foi evidenciado, abrindo diversas possibilidades de desenvolvimento de soluções para variados domínios, tais como: televisão interativa, tele-medicina, educação à distância e videoconferência.

Uma aplicação multimídia manipula diversas mídias com requisitos temporais críticos como áudio e vídeo [1]. O desenvolvimento deste tipo de aplicação apresenta um elevado nível de complexidade, pois precisa tratar diversas questões inerentes à natureza multimídia como a captura, manipulação e exibição de diferentes fluxos de dados e a sincronização entre os elementos da aplicação e o fluxo apresentado.

Contudo, a complexidade das aplicações multimídia não está relacionada apenas aos requisitos impostos pelo aspecto multimídia. Atualmente, existem diversos dispositivos capazes de executar este tipo de aplicação, tais como computadores pessoais, telefones celulares e PDAs [2]. Devido a esta diversidade de dispositivos, há uma grande heterogeneidade de plataformas de execução. As plataformas existentes diferem em uma série de parâmetros, como a capacidade de processamento, armazenamento e comunicação, interfaces para a interação com o usuário, APIs e requisitos de qualidade de serviço (QoS) [3].

Com esta heterogeneidade de dispositivos e interfaces para a interação com o usuário, a implementação destas aplicações torna-se dependente dos tipos de dispositivos e das tecnologias de acesso empregadas [4].

Neste contexto, as plataformas de execução para este tipo de aplicação devem incorporar conceitos de *middleware* [5] de maneira a tratar, de forma transparente, aspectos relacionados a gerenciamento de recursos, controle de acesso, confidencialidade, autenticação, controle de transações e segurança [6]. Isto implica na introdução de uma camada de software entre o sistema operacional e as aplicações.

A complexidade no desenvolvimento de sistemas multimídia não se restringe à heterogeneidade das plataformas de execução. Estes sistemas possuem características que são bastante variáveis, implicando em novos requisitos à medida que novas tecnologias são disponibilizadas. Dessa forma, deve-se permitir ao *middleware* a possibilidade da realização, de forma automática, de ajustes e adaptações entre as interfaces de programação disponibilizadas pelo *middleware* e as possíveis plataformas de hardware e sistemas operacionais [4].

O processo de adaptação possibilita ao software modificar sua estrutura e seu comportamento dinamicamente, em resposta a mudanças em seu ambiente de execução. Segundo Mckinley [7], existem duas abordagens para o processo de adaptação. A primeira ocorre através da adaptação de parâmetros, onde as variáveis dos elementos que formam o sistema são alteradas para determinar o seu comportamento. A segunda abordagem consiste na adaptação composicional, onde a estrutura do sistema pode ser alterada através da substituição, inclusão ou remoção de seus elementos.

O conceito de adaptação está associado aos conceitos de configuração e reconfiguração. Uma configuração trata aspectos relacionados com a estrutura e controle dos elementos do sistema e dos componentes da aplicação, enquanto que uma reconfiguração está associada com as mudanças ocorridas em tempo de execução [4]. Estas mudanças ocorrem em decorrência de variações nas condições do ambiente ou da plataforma de execução, ou devido à evolução do software.

De forma a oferecer o suporte necessário à configuração e reconfiguração, a engenharia de software tem concentrado esforços em 3 tecnologias consideradas como chaves para o desenvolvimento de software reconfigurável: a separação de interesses (*concerns*), a reflexão computacional e o desenvolvimento baseado em componentes [7]. A maioria das abordagens baseia-se na interceptação e redirecionamento das interações entre as entidades do sistema [4][8]. Além disso, estas 3 tecnologias podem ser utilizadas em conjunto.

A separação de interesses permite o desenvolvimento do comportamento funcional da aplicação, isto é, de sua lógica de negócio, de maneira independente dos interesses transversais às aplicações como qualidade de serviço, tolerância à falhas e segurança. Ao separar os interesses transversais do comportamento funcional da aplicação consegue-se simplificar o seu desenvolvimento e sua manutenção, promovendo o reuso de software. A abordagem mais utilizada para o tratamento destas questões é a programação orientada a aspectos [9].

A reflexão computacional pode ser definida como a habilidade de um sistema consultar e alterar seu próprio comportamento [10]. A reflexão permite a um sistema revelar detalhes de sua estrutura e implementação. Assim, pode-se realizar o monitoramento de seus elementos, e a partir deste monitoramento, decidir o que fazer, seja realizando alterações para melhorar o desempenho, para incluir novas capacidades ou para resolver algum problema específico.

O processo de consulta por parte da aplicação sobre o estado do sistema é denominado introspecção, enquanto que o processo de alteração de algum elemento do sistema está relacionado ao conceito de reificação. A introspecção permite ao sistema obter conhecimento das propriedades de seus elementos e ambiente de execução. O conceito de reificação está associado à preservação de informações de compilação dos elementos do sistema de forma a permitir o seu uso dinamicamente [11].

Um sistema reflexivo é dividido em duas partes logicamente distintas: um nível base, onde estão os dados e código responsáveis pela execução da aplicação, e um meta-nível, onde são descritas a estrutura da aplicação e as propriedades dos respectivos elementos de nível base [12]. As alterações nos elementos do nível base do sistema são realizadas de acordo com as descrições nos respectivos meta-elementos. Estes dois níveis estão conectados, fazendo com que as alterações em um deles sejam refletidas no outro.

O protocolo de meta-objetos (*MetaObject Protocol* – MOP) define um conjunto de interfaces com operações que possibilitam a introspecção e a alteração dos meta-elementos da aplicação. Estas operações podem suportar o tratamento de aspectos estruturais como relações de dependência e hierarquia de classes, e o tratamento de aspectos comportamentais, como as interfaces providas por um elemento e os protocolos de comunicação suportados [13].

Segundo Mckinley [7], o desenvolvimento baseado em componentes é a tecnologia de maior impacto no desenvolvimento de software adaptativo dentre as citadas anteriormente. Um componente pode ser definido como uma unidade de software independente, que encapsula seu projeto e implementação e oferece serviços por meio de interfaces bem definidas [14]. Componentes interagem uns com os outros através de interfaces, que são contratos que determinam a forma de comunicação entre componentes.

Em um projeto baseado em componentes, o software é construído através da integração de artefatos reutilizáveis de software pré-existentes desenvolvidos de

maneira independente, que podem ser substituídos ou reconfigurados para satisfazer os requisitos da aplicação. Estes componentes podem ser combinados com outros componentes com o objetivo de produzir um novo componente com um nível mais alto de abstração. Este processo é denominado composição [15].

O desenvolvimento baseado em componentes suporta dois tipos de composição [4]. Em uma composição estática, o desenvolvedor combina diversos componentes em tempo de compilação para produzir uma aplicação. A composição dinâmica permite ao desenvolvedor adicionar, remover ou reconfigurar os componentes de uma aplicação em tempo de execução.

Com isso, a tecnologia de componentes mostra-se como uma das principais alternativas para o desenvolvimento de sistemas multimídia. Ela facilita o desenvolvimento de aplicações distribuídas ao permitir a construção de novas aplicações através da integração de componentes já existentes e garantindo a independência com relação à heterogeneidade das plataformas de execução existentes. Isto é conseguido através das técnicas de composição, o que permite a escolha do componente adequado à plataforma de execução corrente.

Considerando os desafios presentes no desenvolvimento de sistemas multimídia adaptativos, o *framework* Cosmos [3] foi proposto com o objetivo de oferecer uma arquitetura genérica baseada em componentes para a camada de configuração e gerenciamento de recursos em sistemas multimídia distribuídos abertos. O *framework* Cosmos explora o modelo de desenvolvimento baseado em componentes em conjunto com a tecnologia de reflexão computacional, provendo uma arquitetura conceitual para o projeto e o desenvolvimento de plataformas multimídia distribuídas, suportando reflexividade, configuração, e gerenciamento de recursos e componentes.

O Cosmos possibilita a reutilização de componentes que tratam aspectos básicos como a representação e gerenciamento de recursos, propriedades e configuração, permitindo a inclusão, troca ou eliminação de componentes do sistema. Para isso, ele define um conjunto de componentes que fornecem uma visão arquitetural do sistema, podendo ser estendidos, instanciados e customizados para contextos particulares.

O *framework* Cosmos explora o conceito de propriedades, definindo um conjunto de interfaces que dão suporte às tarefas de configuração, gerenciamento e adaptação. Estas interfaces provêem o suporte para a realização de consultas e a definição de características de componentes, consistindo em um mecanismo de suporte à reflexividade.

O Cosmos utiliza o conceito de recurso lógico, denominado *VirtualResource* [16], para representar recursos de hardware e software da plataforma. O conceito de portas de comunicação é explorado para o tratamento de fluxos multimídia. As portas são pontos de conexão por onde os elementos de um fluxo podem entrar ou sair de um componente *VirtualResource*.

As conexões entre os recursos lógicos são gerenciadas por um componente denominado conexão virtual (*VirtualConnection*). Seu papel é abstrair o fluxo de dados multimídia entre as portas provendo uma interface bem definida para a configuração e o gerenciamento das respectivas ligações.

O *framework* define também um modelo de gerenciamento de QoS, suportando o monitoramento do comportamento dos componentes e das conexões entre eles. Este monitoramento é realizado por elementos ativos que notificam ao *middleware* a necessidade de eventuais ajustes nos componentes da aplicação.

O Cosmos define um modelo de meta-programação utilizado para a especificação de componentes e aplicações de maneira independente de tecnologia. O *framework* suporta tanto negociações estáticas quanto dinâmicas para a instanciação, configuração, conexão e aquisição de recursos, suportando também a adaptação dinâmica da aplicação.

1.1. Motivação

As plataformas de *middleware* atuais oferecem transparência de acesso a elementos distribuídos utilizando-se de mecanismos como chamadas de procedimentos remotos (RPC) [17] ou invocação de métodos remotos (RMI) [18]. As interfaces nestes *middlewares* são definidas através de uma linguagem de definição de interfaces (*Interface Definition Language* – IDL) abstraindo os detalhes referentes a uma determinada linguagem de programação. Estas definições são utilizadas para a geração de *stubs* e *skeletons*, que realizam a comunicação entre objetos distribuídos de maneira transparente [19]. Com isso, o desenvolvedor concentra o foco na lógica de negócio da aplicação, abstraindo detalhes referentes a protocolos e mecanismos de comunicação.

Este tipo de interação é baseado na arquitetura cliente-servidor, onde o cliente envia requisições para um servidor que processa esta requisição e envia uma resposta para o cliente. Esta comunicação entre cliente e servidor concentra-se na transmissão de

fluxos de controle através dos *stubs* e *skeletons*, utilizando um protocolo de comunicação padrão.

Entretanto, esta abordagem não é satisfatória para sistemas multimídia, que são centrados na transmissão de fluxos de informação [20]. Isto acontece por que os mecanismos de comunicação destas plataformas foram projetados visando à confiabilidade da comunicação, enquanto que, para sistemas multimídia, os requisitos de tempo-real são mais importantes para o tratamento adequado de um fluxo de informação. Em um fluxo de informação a perda de alguns dados é aceitável de forma a atender as restrições de tempo destes sistemas.

A interface utilizada pelos clientes constitui uma referência remota a um objeto ou serviço. Esta referência, denominada *binding*, encarrega-se dos detalhes referentes à comunicação entre os objetos [19]. Existem duas abordagens para o estabelecimento de *bindings*. Um *binding* implícito é aquele criado e configurado de maneira automática pelo *middleware*, enquanto que um *binding* explícito permite a seleção e configuração de parâmetros específicos ao tipo de *binding* selecionado. Um *binding* explícito é representado através de um objeto [21] permitindo a sua manipulação e configuração por parte da aplicação.

A maioria dos sistemas de *middleware* tradicionais utiliza-se da abordagem de *binding* implícito, onde os mecanismos não são expostos para as aplicações e não podem ser consultados, configurados ou estendidos. Esta abordagem de ocultar os detalhes referentes à comunicação permite que o desenvolvedor mantenha seu foco na lógica de negócio de sua aplicação, e não em detalhes relacionados a protocolos e mecanismos de comunicação.

Entretanto, a transmissão de fluxo de dados multimídia frequentemente requer a otimização dos parâmetros de rede, como o tamanho dos *buffers* de envio e recebimento. Em sistemas multimídia os programadores precisam lidar com propriedades do fluxo relacionadas à QoS e o comportamento dos mecanismos de suporte à comunicação não dependem somente do tipo de informação que é transmitida, mas das características do fluxo atual.

Além disso, este tipo de aplicação precisa se adaptar aos requisitos impostos por seus usuários e ao seu ambiente de execução. Estes requisitos podem ser alterados em tempo de execução, fazendo com que as aplicações sejam alteradas dinamicamente de forma a atender aos novos requisitos [22]. O mecanismo de *binding* implícito não

permite o monitoramento de parâmetros de QoS, o que dificulta a identificação de situações onde uma adaptação seja necessária.

Neste sentido, plataformas de *middleware* multimídia precisam oferecer mecanismos de comunicação específicos para este tipo de aplicação de forma a suportar a troca de fluxos de dados multimídia entre seus elementos. Estes mecanismos devem prover um alto nível de flexibilidade no tratamento de QoS devido às características do fluxo e a necessidade de adaptação.

Um fluxo de informação constitui um *pipeline* conectando produtores e consumidores de informação, possuindo diversos requisitos não-funcionais e requisitos de tempo real críticos. Além disso, segundo Koster [20], a utilização de um protocolo padrão de comunicação voltado para a transmissão de fluxos de controle não oferece o suporte necessário à variedade de requisitos impostos pela transmissão de fluxos de informação, o que evidência a necessidade de mecanismos de comunicação específicos para este tipo de aplicação.

O Cosmos é um *framework* genérico baseado em componentes, que explora o conceito de reflexividade, dando suporte a configuração e reconfiguração dinâmica. Com isso, faz-se necessário a definição de um mecanismo de interconexão voltado para a transmissão de fluxos de informações e que suporte os conceitos utilizados pelo *framework* Cosmos, tais como a configuração e reconfiguração dinâmica de seus elementos além do suporte a reflexão computacional e à tecnologia de desenvolvimento baseado em componentes.

1.2. Objetivos

O objetivo principal deste trabalho é a concepção, projeto e implementação de um mecanismo de comunicação específico para aplicações multimídia distribuídas que atenda aos requisitos impostos por este tipo de aplicação e que possa ser incorporado ao *framework* Cosmos. Para atender essas necessidades, definiu-se um modelo de interconexão para ambientes multimídia utilizando os conceitos da tecnologia de desenvolvimento baseado em componentes. Esse modelo oferece um mecanismo flexível, possibilitando a integração de uma variedade de tipos de componentes em ambientes distribuídos e heterogêneos.

O modelo explora os conceitos de porta, conexão virtual e canal de comunicação. Estes conceitos são utilizados na definição de uma arquitetura de

interconexão de componentes, proposta neste trabalho, para ambientes multimídia distribuídos abertos. A arquitetura abstrai os detalhes dos mecanismos de comunicação empregados e oferece um conjunto de interfaces para o acesso às propriedades destes mecanismos, dando suporte à reconfiguração dinâmica.

Este trabalho está inserido no contexto do *framework* Cosmos. O Cosmos define uma série de conceitos com o objetivo de facilitar o desenvolvimento de sistemas multimídia distribuídos. Através da utilização da tecnologia de componentes o *framework* provê um grande nível de abstração e independência entre plataformas, linguagens e protocolos, entre outros. O Cosmos provê um alto nível de flexibilidade ao permitir que *middlewares* específicos baseados no *framework* tratem e especializem cada aspecto de maneira independente. O *framework* permite também o reuso de componentes que tratam aspectos básicos relacionados com a representação e o gerenciamento de recursos. Por se basear no modelo de desenvolvimento baseado em componentes, o *framework* permite introduzir, trocar ou eliminar componentes do sistema através da análise das interfaces requisitadas e do universo de componentes capazes de oferecer estas interfaces.

O conceito de porta constitui-se na principal abstração para a realização de comunicação entre componentes, servindo como interface de comunicação para o envio e o recebimento de dados. Através da utilização das portas de comunicação os detalhes relacionados à troca de informações entre os componentes são tratados de maneira transparente.

As portas de comunicação são tratadas como componentes, possuindo uma interface de propriedades, definida pelo *framework* Cosmos, que permite a configuração de seus parâmetros. Os componentes que fazem parte desta arquitetura de interconexão também possuem interfaces que são utilizadas pelos elementos do *middleware* para sua configuração.

Como base para esta abstração, foi utilizado o conceito de canal de comunicação [23][24] para realizar a interligação entre as portas de comunicação. O canal de comunicação é gerenciado pela conexão virtual, oferecendo o suporte necessário à adaptação dinâmica dos componentes da aplicação.

O Cosmos define mecanismos de negociação de propriedades e adaptação com o objetivo de oferecer o suporte necessário à construção de sistemas adaptativos e configuráveis. Sistemas de *middleware* baseados no Cosmos atuam de forma autônoma, negociando e modificando a composição do software em tempo de execução para

atender as necessidades impostas pelas aplicações. De maneira a possibilitar a reconfiguração da aplicação, através da troca de componentes ou da mudança de parâmetros operacionais dos componentes envolvidos, é necessária a definição de mecanismos que permitam a adaptação das aplicações sem colocá-las em situações de risco ou interrupção.

A utilização do canal de comunicação provê transparência de comunicação entre as portas em relação a detalhes como a localização dos componentes e o protocolo de comunicação utilizado, além de oferecer o suporte necessário para que a reconfiguração dinâmica não interfira nos componentes da aplicação.

Para mostrar a viabilidade do modelo de interconexão, os conceitos introduzidos foram implementados resultando em uma extensão de um protótipo de um *middleware* para sistemas de televisão digital interativa denominado AdapTV. Esta extensão foi realizada de forma a incorporar os conceitos e elementos do Cosmos e do modelo de interconexão definido. Neste trabalho foram realizados testes com a instanciação de uma interconexão e sua adaptação em um ambiente local e, posteriormente, em um ambiente distribuído. Esta instanciação envolveu a definição de uma aplicação exemplo para o *middleware* AdapTV, envolvendo a captura, transmissão e apresentação de fluxo multimídia em uma plataforma distribuída.

1.3. Organização do Trabalho

O restante deste trabalho está organizado da seguinte maneira: o Capítulo 2 apresenta o *framework* Cosmos e seus principais elementos, de maneira a contextualizar o modelo de interconexão definido. O Capítulo 3 apresenta o modelo de interconexão de componentes proposto, enquanto que o Capítulo 4 descreve a implementação realizada de maneira a realizar uma prova de conceito dos elementos introduzidos pelo trabalho. O Capítulo 5 apresenta uma breve discussão envolvendo alguns trabalhos relacionados, fazendo uma comparação em relação a presente proposta. O Capitulo 6 apresenta as conclusões e algumas proposições de trabalhos futuros.

2. O Framework Cosmos

Este capítulo apresenta o *framework* Cosmos – um *framework* genérico baseado em componentes para o projeto e o desenvolvimento da camada de *middleware* para uma variedade de sistemas multimídia distribuídos – com o objetivo de contextualizar o modelo de interconexão definido.

O Cosmos foi proposto para dar suporte à configuração e gerenciamento de recursos e componentes de aplicações de sistemas multimídia distribuídos. Para isso, o Cosmos define um modelo de componentes abstrato que incorpora as funcionalidades básicas dos elementos arquiteturais do sistema, podendo ser estendido, instanciado e especializado para contextos particulares. O Cosmos define também um modelo de recursos virtuais e um modelo de gerenciamento de QoS.

O modelo de interconexão foi proposto para tratar os aspectos envolvidos com a comunicação entre componentes multimídia, e para fins de prova de conceito foi aplicado no contexto do *framework* Cosmos.

Uma instanciação do Cosmos foi realizada como prova de conceito para o *framework*. Esta instanciação resultou na definição de um *middleware* adaptativo para sistemas de televisão digital interativa denominado AdapTV [25]. O AdapTV envolve os principais elementos propostos pelo *framework* Cosmos.

Neste capítulo, são apresentadas as principais características do *framework*, envolvendo os elementos, conceitos, modelo de componentes e sua arquitetura.

2.1. Modelo de Componentes e conceitos principais

O Cosmos concentra seus esforços na representação e manipulação da diversidade de conceitos, requisitos e componentes relacionados com os sistemas multimídia. O *framework* definido consiste de um conjunto de componentes abstratos que podem ser estendidos, instanciados e customizados para contextos particulares, fornecendo uma visão arquitetural do sistema onde são abstraídos os detalhes de implementação desses componentes.

Com o objetivo de tratar os aspectos de generalidade e reusabilidade, o *framework* define um conjunto de interfaces e regras para gerenciamento e interação entre os diversos componentes do sistema. Como base para esta abordagem, o Cosmos define um componente abstrato, denominado *CosmosComponent*, com as interfaces básicas que devem ser providas obrigatoriamente por qualquer componente concreto do *framework*. A Figura 1 apresenta um modelo simplificado que descreve os principais elementos arquiteturais do Cosmos e seus relacionamentos.

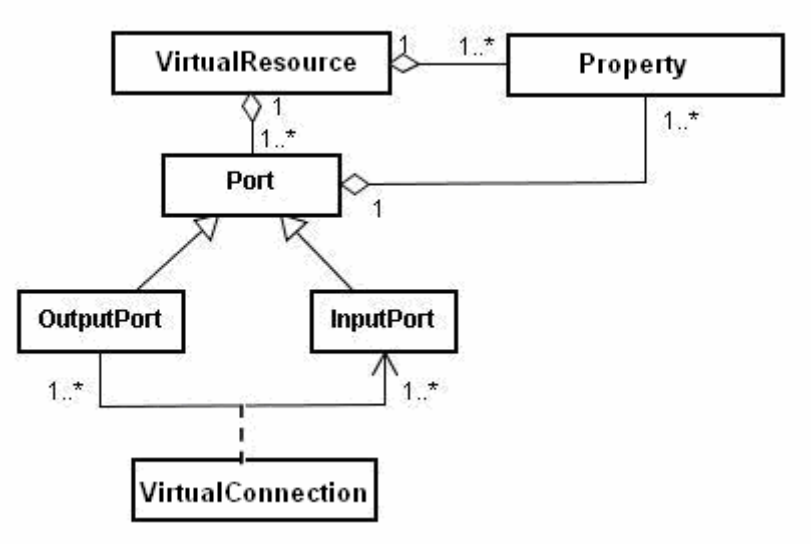

Figura 1. Principais elementos arquiteturais do Cosmos[4].

O conceito de recurso virtual, denominado *VirtualResource*, constitui-se numa das características fundamentais do Cosmos. *VirtualResources* são componentes que efetivamente representam e gerenciam os elementos associados ao processamento de um fluxo de mídia, provendo abstrações para a representação de recursos de hardware ou software da plataforma. Além de implementar as interfaces básicas definidas pelo *framework*, eles podem definir outras interfaces, de modo a customizá-los de acordo com as funcionalidades do recurso efetivo que ele representa [26].

Um *VirtualResource* pode ter várias portas de comunicação, e cada porta tem um tipo associado que descreve as características do fluxo de dados que passa por ela. O *framework* Cosmos define diferentes tipos de interfaces a serem oferecidas pelas portas, cada uma com semântica distinta, como por exemplo, interfaces operacionais utilizadas pelos *VirtualResources*, caracterizando a porta como sendo do tipo porta de saída (*OutputPort*) ou porta de entrada (*InputPort*).

Portas de saída implementam interfaces que provêem suporte para comunicações que fluem do *VirtualResource*, enquanto que portas de entrada recebem fluxos destinados ao *VirtualResource*. No modelo simplificado da Figura 1 foi introduzido um elemento *Port* para tratar os conceitos e comportamentos comuns às portas. As portas também são componentes, devendo possuir, além de suas interfaces operacionais, as interfaces básicas definidas pelo *framework* que são utilizadas para fins de configuração, permitindo a definição, monitoramento ou alteração dinâmica de propriedades associadas às mesmas. Os tipos concretos de componentes *OutputPort* e *InputPort* definem respectivamente comportamentos para portas de saída e de entrada.

Os componentes locais ou remotos são interconectados através de ligações entre portas de comunicação. Estas ligações são gerenciadas por uma conexão virtual, denominada *VirtualConnection*. A conexão virtual provê uma interface bem definida para configuração e gerenciamento das ligações entre as portas, abstraindo os detalhes inerentes ao fluxo de dados multimídia. Um componente *VirtualConnection* não realiza a transferência dos dados do fluxo, seu papel é criar, configurar e gerenciar os recursos e componentes do *middleware* que realizam a comunicação. A configuração e o gerenciamento dos componentes que compõem os mecanismos de comunicação acontece de forma transparente pelo *VirtualConnection*.

O Cosmos define um modelo de componentes próprio, com o objetivo de prover um modelo neutro que trate, de maneira independente, a diversidade de tecnologias existentes. Para tal, definiram-se um conjunto de interfaces, denominadas interfaces básicas, que são providas por todos os componentes do modelo. Dentre as interfaces deste conjunto encontram-se as interfaces de propriedades.

Estas interfaces incorporam os conceitos de propriedades definidos e explorados em [3], provendo características de flexibilidade e adaptabilidade. Uma propriedade pode ser definida dinamicamente como um par de elementos (chave, valor), no qual 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.

Explorando estes conceitos, diversos dispositivos podem ser configurados e customizados de maneira uniforme através da definição de valores específicos de propriedades, consistindo num mecanismo de suporte genérico para o gerenciamento de recursos e adaptação. Além das interfaces de propriedades, as interfaces básicas contemplam operações que permitem o gerenciamento de recursos e ciclo de vida.

As interfaces de propriedades dão suporte às tarefas de configuração, gerenciamento e adaptação, definindo aspectos não-funcionais de componentes. Estas interfaces consistem em um mecanismo de suporte ao conceito de reflexividade, possibilitando a realização de consultas e definição de características de componentes. Desse modo, os componentes da aplicação podem consultar e definir mudanças no comportamento do componente através de operações das interfaces de propriedades.

As operações básicas definidas para a manipulação de propriedades foram agrupadas em diferentes interfaces do componente abstrato *CosmosComponent*, conforme ilustrado na Figura 2. Desta maneira, todos os componentes do *framework* implementam o conjunto de interfaces básicas que são utilizadas pelo *middleware* para fins de gerenciamento.

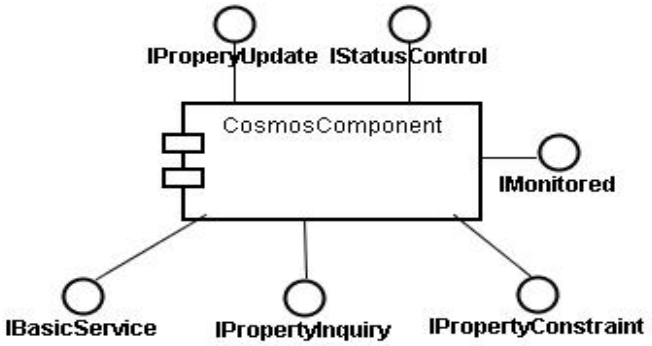

Figura 2. Interfaces básicas do Componente *CosmosComponent.*

A interface *IBasicService* fornece operações para obtenção de informações básicas do componente, como por exemplo, as interfaces por ele providas. A interface *IPropertyInquiry* é definida para consultas de valores de propriedades. A interface *IPropertyConstraint*, fornece operações para a restrição de valores para efeito de 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.

As interfaces de propriedades que foram introduzidas no componente básico *CosmosComponent* permitem a realização de mudanças nas propriedades dos componentes. Para permitir mudanças consistentes de configuração, o *middleware* deve supervisionar as mudanças verificando se a mudança é coerente com o estado atual da plataforma.

A interface *IStatusControl* oferece uma operação para gerenciamento e controle de mudança de estado do componente. Cada componente tem uma máquina de estados associada que determina os possíveis comportamentos e transições de estado válidas. A interface *IPropertyUpdate* é utilizada para a atualização de valores selecionados de propriedades dos componentes quando houver eventuais alterações em decorrência, por exemplo, de negociações dinâmicas, e para a ativação e desativação do componente.

Outro aspecto importante do *framework* é o suporte ao gerenciamento de QoS [4]. 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. A interface *IMonitored* oferece suporte ao modelo de QoS definido pelo *framework*, que especifica um conjunto de operações que permitem o gerenciamento de QoS.

2.2. *Framework* Arquitetural de Alto-Nível

O Cosmos dá suporte às tarefas de configuração e gerenciamento de recursos e componentes. Para isso ele define um *framework* arquitetural de alto-nível apresentado na Figura 3.

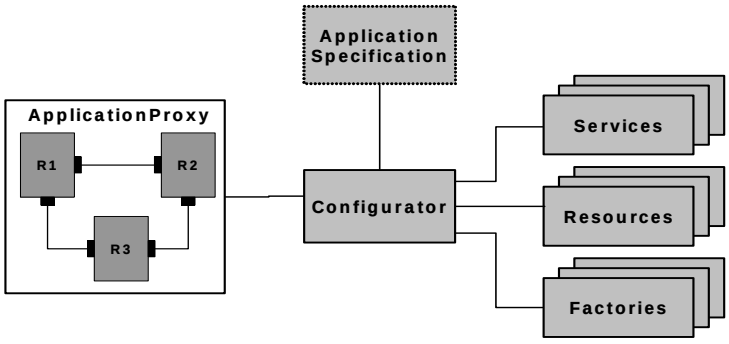

Figura 3. Arquitetura de Alto Nível do Cosmos

Neste *framework* arquitetural, os seguintes elementos são destacados:

- ***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 uma descrição dos componentes de uma aplicação em execução, sendo responsável pelas operações de manipulação de propriedades desses componentes;
- ***Factories*:** responsáveis pela criação de componentes (portas, serviços, recursos e demais componentes do *framework*);
- ***Resources*:** representação de recursos (*hardware* e *software*); e
- ***Services*:** permite agregar novas funcionalidades a sistemas de *middleware* implementados.

Como elemento central desta arquitetura, o Cosmos define o *Configurator*, cujo papel consiste em coordenar e gerenciar todas as fases do ciclo de vida dos componentes do *middleware* e da aplicação. O *Configurator* é um componente crítico, sendo responsável pela iniciação, configuração e gerenciamento dos recursos do sistema, dos componentes do *middleware* e da aplicação. Este componente realiza operações de negociação e ajuste dinâmico de propriedades, que estão associadas com os recursos e fluxos multimídia envolvidos no sistema.

Uma aplicação no Cosmos deve ser definida através de uma especificação (*Application Specification*), que descreve os recursos e os elementos da arquitetura envolvidos, assim como os vários requisitos de QoS necessários para a sua execução. Esta especificação é realizada usando uma linguagem de configuração, que possui uma estrutura similar a uma ADL [27] (*Architecture Description Language*).

O Cosmos não define uma tecnologia, nem tampouco uma linguagem específica para a descrição de aplicações e componentes. No entanto, o *Configurator* precisa interpretar especificações de modo a poder construir e representar as respectivas aplicações. Para isso, o *framework* define um modelo de meta-componentes, apresentado em [28], que serve de base para a descrição e representação de aplicações no *middleware*. Dessa forma, a definição da linguagem a ser utilizada é de responsabilidade do *middleware* que implementa o *framework*, devendo esta estar baseada no modelo de meta-componentes definido. Isso possibilita que a configuração dos componentes e aplicações possa ser gerada e processada pelo *middleware* independentemente de que linguagem venha ser utilizada em uma instanciação do *framework* Cosmos.

A descrição de cada aplicação, conforme o modelo definido no Cosmos, é representada no *middleware* por um componente denominado *ApplicationProxy*. É através do componente *ApplicationProxy* que o Cosmos dá suporte ao conceito de reflexividade. Este componente atua como um repositório em um meta-nível, onde são descritas a estrutura da aplicação e as propriedades dos respectivos componentes de nível base.

A representação da aplicação no *ApplicationProxy* é realizada através de um grafo com os meta-componentes que representam os componentes do sistema e da aplicação. Estes meta-componentes funcionam como repositórios de meta-dados (propriedades) associados aos recursos, descrevendo as propriedades e os estados dos componentes associados, e dando suporte ao gerenciamento, configuração e adaptação [2].

O *ApplicationProxy* facilita o suporte à consistência em processos de adaptação. Cada operação associada a uma mudança dinâmica de propriedades de um componente deve ser encaminhada ao componente *ApplicationProxy* que verifica a coerência desta operação, atualiza a descrição da arquitetura e repassa estas alterações para os respectivos componentes afetados.

2.3. Modelo de Meta-dados e a Linguagem de Descrição Arquitetural

Como mencionado, o Cosmos define que aplicações e componentes devem ser descritos através de uma linguagem de especificação. O *Configurator* é o componente responsável por processar esta especificação e realizar acordos envolvendo a negociação de propriedades entre os componentes. 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.

Neste sentido, o *framework* não define uma linguagem específica para a descrição de aplicações e componentes, mas define um meta-modelo de descrição e representação de aplicações. Este meta-modelo contempla os conceitos e elementos do *framework*, e é utilizado pelo *Configurator* para preparar, configurar e controlar a execução de aplicações.

Uma descrição deste modelo de meta-dados é apresentada na Figura 4. Os elementos descritos na figura contemplam os componentes do modelo arquitetural do

framework Cosmos. Estes meta-componentes descrevem as características de seus respectivos componentes.

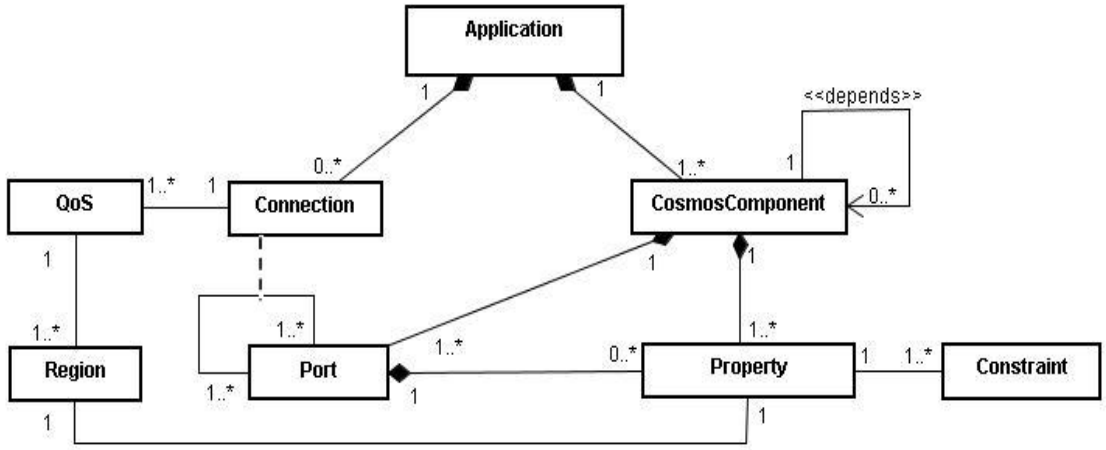

Figura 4. Modelo de meta-dados do *framework* Cosmos.

Uma aplicação é representada pelo componente *Application*, que pode ser composto por um ou mais componentes (*CosmosComponent*) e por nenhum ou muitos *Connection*, elemento que representa uma conexão.

A descrição de um componente (*CosmosComponent*) pode definir uma ou várias propriedades (*Property*) e nenhuma ou várias portas (*Port*). Podem existir também restrições (*Constraint*) associadas a propriedades dos componentes. Componentes podem ser constituídos por outros componentes, suportando o conceito de composição, através de uma relação de dependência.

As portas (*Port*), que também são tratadas como componentes, também possuem propriedades e restrições associadas a estas propriedades. Conexões, representadas por componentes *Connection*, descrevem o relacionamento entre portas, devendo existir pelo menos uma porta de origem e uma de destino. Além disso, as conexões podem conter componentes que descrevem requisitos de QoS (*QoS*), que são utilizados para gerenciamento de QoS seguindo o modelo de gerenciamento de QoS definido pelo *framework*. O modelo de QoS define regiões de adaptação (faixa de valores de propriedades) que são representados pelo componente *Region*.

A especificação de uma aplicação, envolvendo a descrição dos componentes, propriedades, suas portas e respectivas conexões, deve ser processada por um *parser* de acordo com a sintaxe da linguagem de descrição definida. A Figura 5 ilustra este

processo. O *parser* tem o papel de mapear a descrição da aplicação em estruturas internas denominadas meta-dados.

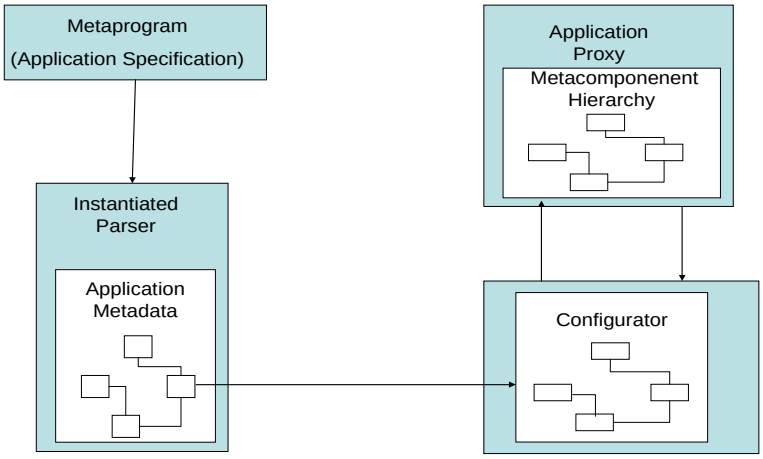

Figura 5. Esquema de processamento de uma especificação.

Estes meta-dados são dados que descrevem as características dos componentes da aplicação e da própria plataforma. Eles são utilizados pelo *Configurator* para a realização de negociações entre os componentes da aplicação, e ficam armazenados no *ApplicationProxy* na forma de um grafo de meta-componentes.

O princípio desta abordagem consiste na abstração da escolha de uma linguagem de descrição específica, permitindo a introdução de novas linguagens, sem que isto venha a trazer impactos na implementação do componente *Configurator*. Desta maneira, o desenvolvedor tem a possibilidade de utilizar as linguagens de descrição que ele escolher ou projetar, devendo definir seus respectivos *parsers* que deverão mapear as estruturas da linguagem para o modelo de meta-dados do Cosmos.

A definição de uma linguagem de especificação é de responsabilidade do *middleware* que implementa o *framework*. Esta linguagem deve respeitar o meta-modelo definido de forma a gerar a hierarquia de meta-dados a ser reconhecida e processada pelo *Configurator*. Para isso, o meta-modelo definido pelo Cosmos é expresso utilizando-se uma notação baseada em EBNF em [29], servindo de base para a definição de novas linguagens.

Com base no meta-modelo definido, foi definida uma linguagem de especificação de aplicações, baseada em XML, para o protótipo do *middleware* AdapTV. Com o objetivo de dar uma visão geral das características do modelo de

descrição, alguns elementos da especificação dos componentes de uma aplicação desenvolvida no AdapTV são apresentados.

A Figura 6 mostra a especificação XML de um componente denominado *VideoFlowProducer*. Este é um componente produtor de vídeo que possui uma porta de saída associada, e foi utilizado durante os testes do *middleware* AdapTV. Na especificação são explicitadas as capacidades e propriedades do fluxo produzido pelo componente, como a taxa (propriedade *framerate*) e o padrão de codificação (propriedade *encoding*).

```
<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>
```

Figura 6. Descrição XML do Componente *VideoFlowProducer*.

A Figura 7 apresenta um trecho da especificação de uma configuração exemplo para uma aplicação, denominada *VideoFlowApp*. Esta aplicação é formada por dois componentes, um componente produtor, cujo papel é distribuir vídeos, e um componente consumidor para receber e exibir os mesmos. 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. O produtor tem capacidade para oferecer vídeos com diferentes níveis de QoS, onde cada vídeo pode ser transmitido em várias taxas (*frames* por segundo).

Seguindo o modelo de gerenciamento de QoS definido pelo Cosmos, a especificação de parâmetros de QoS é realizado através de regiões, que são delimitadas por faixas de valores (*range*) e um valor *default* para cada região.

```
<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>
```

Figura 7. Descrição XML da configuração da aplicação *VideoFlowApp*.

No exemplo apresentado, 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.

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. A próxima seção apresenta uma breve explicação sobre o funcionamento dos elementos definidos pelo *framework* Cosmos.

2.4. Arquitetura Funcional

O *Configurator* utiliza o grafo de meta-dados gerado pelo *parser* para executar as operações de configuração. A configuração consiste na consulta e processamento dos meta-dados deste grafo, de modo a identificar quais recursos deverão ser instanciados e suas respectivas propriedades, as quais determinam valores para os parâmetros de configuração destes recursos.

O *Configurator* é o elemento que gerencia os componentes instanciados em seu espaço de endereçamento. A instanciação de um recurso virtual é realizada por uma fábrica acionada pelo *Configurator*. Caso este recurso precise ser instanciado em outro espaço de endereçamento, o *Configurator* deverá se comunicar com o *Configurator* remoto responsável por gerenciar os recursos deste outro espaço de endereçamento, que irá acionar a fábrica correspondente a partir dos meta-dados definidos. Estes meta-dados residem no *ApplicationProxy*, que se encontra no mesmo espaço de endereçamento que o *Configurator* que processou a especificação. Este *Configurator* é denominado como *Configurator* mestre, enquanto que os demais *Configurators* envolvidos remotamente são denominados *Configurators* escravos.

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 meta-componentes representado e manipulado por um componente *ApplicationProxy*, sob o controle e supervisão do respectivo *Configurator* mestre. O *ApplicationProxy* mantém referências simbólicas para os recursos virtuais remotos, provendo o suporte para a realização de *binds* permitindo ao *Configurator* mestre acessar esses recursos remotos.

Os *Configurators*, mestre e escravos, interagem entre si através de um protocolo de comunicação entre configuradores baseado em RPC. Para tal, o Cosmos define uma interface padrão para a comunicação entre configuradores. Esta interação envolve questões sobre a alocação de recursos, negociação de propriedades e instanciação dos componentes remotos.

O nível de interação entre *Configurators* remotos limita-se 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). 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 configuração.

Apesar do *Configurator* mestre 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 consultar este *Configurator* remoto de modo a verificar se a configuração indicada é coerente com a disponibilidade de recursos da plataforma local.

O componente *ApplicationProxy* dá suporte para as características de reflexividade e adaptação, realizadas de maneira automática pelo *Configurator*. O *framework* suporta a realização de adaptações originadas por necessidades de evolução do sistema, por decisão da aplicação, denominada de adaptação reativa, ou por decisão interna do próprio *middleware* ao detectar alguma anomalia que venha a degradar o desempenho do sistema. Este tipo de adaptação é denominado como adaptação pró-ativa.

Para a realização de uma adaptação, o *ApplicationProxy* é consultado acerca do estado atual do sistema, tanto pelo *middleware* quanto pela aplicação. De posse dessas informações, são realizadas operações de verificação para identificar possíveis valores comuns de propriedades. 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.

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 determina o momento adequado para mudar a configuração.

As alterações de valores de propriedades são realizadas através das interfaces oferecidas pelo *ApplicationProxy*, porém somente o *Configurator* tem acesso a estas interfaces, fazendo com que os componentes da aplicação tenham que se reportar ao *Configurator* para a modificação de valores de propriedades. Desta maneira, mesmo que seja possível à aplicação obter informações acerca do estado atual do sistema, a

mudança de configurações só é realizada pelo *Configurator*, garantindo a consistência das modificações.

De acordo com a arquitetura definida, os recursos requeridos associados a uma instância de um recurso virtual só serão efetivamente alocados após a verificação por parte do *Configurator* acerca da consistência da configuração. Esta verificação só é possível após o grafo da aplicação estar montado, fornecendo uma visão global dos recursos requeridos. Estas informações acerca da estrutura da aplicação são mantidas armazenadas no *ApplicationProxy*.

A Figura 8 apresenta a arquitetura definida pelo Cosmos para dar suporte à execução de várias aplicações. Para cada aplicação o *Configurator* cria um componente *ApplicationProxy*. Durante o processo de configuração, o *Configurator* interage com o *ApplicationProxy* de cada aplicação para incluir ou recuperar informações ou propriedades de recursos virtuais.

Uma adaptação originada pela aplicação deve ocorrer através de operações na API do usuário, fazendo com que a mudança na configuração seja realizada sob o controle do *Configurator*, que verifica a consistência da modificação e realiza as devidas alterações junto ao *ApplicationProxy*. Uma aplicação utiliza a API de reflexividade para obter informações acerca do estado atual do sistema.

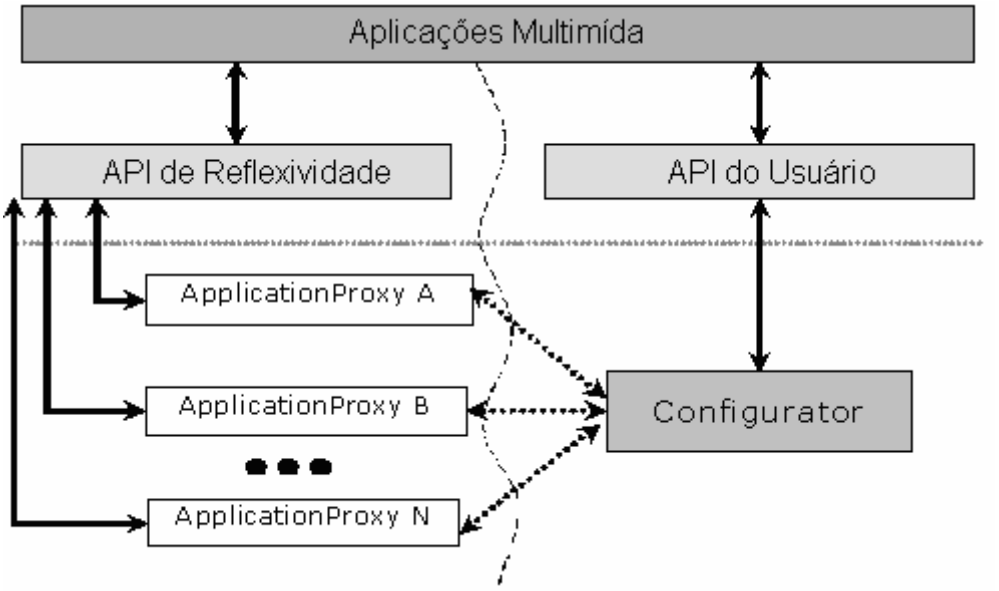

Figura 8. Configuração de Aplicações.

Uma adaptação que ocorre em resposta a uma ação da aplicação é denominada adaptação reativa. Neste caso, a aplicação utiliza a API de usuário para a realização de

alterações de propriedades. Estas alterações são realizadas após a análise por parte do *Configurator* da consistência das modificações. Este controle é necessário, pois interações do usuário podem alterar os requisitos da aplicação, ou do sistema, fazendo com que uma alteração no ambiente de execução de uma aplicação possa provocar alterações no ambiente das outras.

Por sua vez, o *Configurator* pode tomar a iniciativa de realizar a adaptação. Esta abordagem é denominada adaptação pró-ativa, onde o *middleware* realiza a adaptação segundo a orientação estabelecida na especificação da aplicação.

Para atender aos requisitos de reconfiguração dinâmica, o *framework* provê os conceitos de reflexividade, cujos mecanismos de suporte são providos pelas interfaces de configuração do componente *ApplicationProxy* e gerenciadas pelo *Configurator*.

Em uma adaptação pró-ativa, o *middleware* acessa diretamente as interfaces de configuração do componente *ApplicationProxy*. Este tipo de adaptação envolve um modelo de gerenciamento de QoS definido pelo Cosmos. O modelo de QoS define um elemento de observação, denominado monitor, que é responsável por notificar possíveis violações de acordos de QoS. Neste caso, o monitor notifica a ocorrência da violação de um acordo e o *Configurator* realiza a adaptação.

2.5. Modelo de Gerenciamento de QoS

Como mencionado anteriormente, o Cosmos define elementos de observação responsáveis por notificar possíveis violações de acordos de QoS. Estes elementos são denominados monitores, sendo responsáveis por verificarem dinamicamente se os parâmetros de QoS do sistema se comportam de acordo com os requisitos da aplicação. Estes monitores são componentes arquiteturais ativos definidos pelo modelo de gerenciamento de QoS apresentado em [30].

O modelo utiliza o conceito de intervalos de QoS, no qual a definição dos requisitos ocorre por faixas de valores. Além disso, o modelo permite também a definição de um valor default, servindo como referência para dar início à operação dos recursos nas respectivas regiões. 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.

O modelo de gerenciamento de QoS define um conjunto de componentes que fornecem as funcionalidades necessárias para provisão de QoS. Os gerentes de QoS

(componente *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.

Após a instanciação de um gerente de QoS, o *Configurator* define os meta-dados 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 meta-dados 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 meta-componentes do *ApplicationProxy*. Feita a análise, o *QoSManager* repassa a notificação ao *Configurator*, que coordena o processo de adaptação.

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 *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 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.

2.6. Considerações Finais

Este capítulo apresentou uma visão geral dos principais conceitos e características do *framework* Cosmos. O Cosmos é um *framework* genérico baseado em

componentes para a camada de *middleware* para uma variedade de sistemas multimídia distribuídos [4]. O modelo de interconexão proposto está sendo testado no contexto do *framework* Cosmos, provendo um mecanismo de comunicação entre componentes para a troca de fluxos de dados multimídia, oferecendo o suporte à reconfiguração dinâmica.

A abordagem arquitetural do *framework* definido oferece níveis elevados de flexibilidade, provendo suporte a características de interoperabilidade, QoS e adaptabilidade.

3. Modelo de Interconexão de Componentes

Este capítulo apresenta o modelo de interconexão proposto e seus principais componentes. O modelo foi definido baseado na necessidade de um mecanismo de comunicação específico para sistemas multimídia distribuídos e nos conceitos utilizados pelo *framework* Cosmos. O objetivo é definir um mecanismo de comunicação genérico que ofereça o suporte necessário à adaptação e reconfiguração dinâmica em sistemas multimídia distribuídos e como prova de conceito, aplicar este mecanismo no contexto do *framework* Cosmos. Inicialmente, suas principais características são apresentadas, seguido dos componentes que formam o modelo.

3.1. Conceitos Principais

Os componentes do modelo definido pelo *framework* Cosmos interagem uns com os outros, como origem ou destino de dados, através de portas de comunicação, utilizando-as para a troca de fluxo de dados multimídia. O *framework* define também o conceito de conexão virtual, com o objetivo de representar e gerenciar as conexões estabelecidas. Uma conexão virtual não é utilizada para a troca de dados efetivamente. Seu papel é servir de ponto de acesso para as operações de gerenciamento dos elementos que compõem uma interconexão, permitindo a abstração dos detalhes referentes à topologia de interconexão, QoS, sincronização e protocolos de comunicação. As portas de comunicação são registradas em uma conexão virtual que se responsabiliza por seus respectivos gerenciamentos.

A porta de comunicação é uma abstração que define um ponto de interação de um componente. Ela oferece uma interface padrão, independente das tecnologias de comunicação suportadas pela plataforma onde o componente venha a ser instanciado. De forma a abstrair os detalhes destas tecnologias, foi explorado o conceito de canal de comunicação [23][24]. O canal de comunicação realiza a ligação entre duas portas, sendo utilizado pelas mesmas para a troca de dados.

A Figura 9 apresenta o relacionamento entre os recursos virtuais, portas de comunicação, canal e conexão virtual. As portas de comunicação são acessadas pelos

recursos virtuais e registradas junto à conexão virtual. Por sua vez, a conexão virtual se encarrega de gerenciar os elementos da interconexão, criando, por exemplo, o(s) canal(ais) de comunicação necessário(s) para a interligação entre as portas. A conexão virtual também serve como ponto de acesso do configurador aos elementos envolvidos.

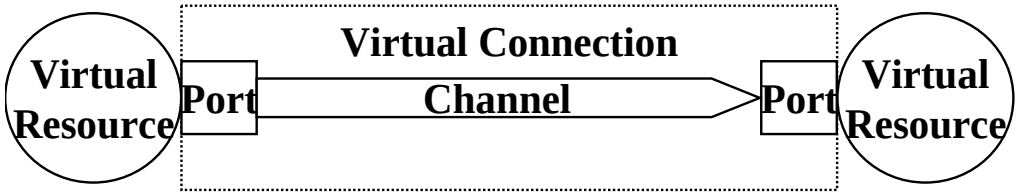

Figura 9. Relacionamento entre os elementos do modelo.

A utilização do canal de comunicação para estabelecer ligações entre portas permite ao *middleware* escolher diferentes tecnologias e protocolos de comunicação. Com isso, a escolha da tecnologia de comunicação a ser empregada na ligação entre duas portas pode levar em consideração o tipo de dado a ser transmitido, parâmetros de rede e a localização dos respectivos componentes participantes (recursos virtuais).

O modelo suporta diversas topologias de interconexão, permitindo conexões 1x1, 1xN, Nx1 e NxN. As portas de comunicação podem funcionar como portas de entrada ou portas de saída. A conexão entre uma porta de saída e várias portas de entrada caracterizam uma topologia 1xN. O componente *VirtualConnection* representa uma conexão entre duas ou mais portas; a definição do número de portas envolvidas depende das ligações estabelecidas pela especificação da aplicação.

A porta de comunicação opera com dados em formato de seqüências de bytes. Cada porta de comunicação possui um conjunto de propriedades que define os parâmetros do fluxo associado, ou seja, que transita por ela. Isto acontece devido ao fato de um recurso poder possuir diversas portas de comunicação, podendo ser portas de entrada ou portas de saída, com possibilidades de tratar diferentes formatos, como acontece, por exemplo, com componentes conversores de fluxos de um formato para outro.

Esta abordagem adiciona um nível de flexibilidade ao modelo, uma vez que as portas de comunicação não precisam conhecer os dados que passam por elas. Com isso, o modelo pode ser facilmente aplicado a outros *middlewares* sem impactos para a interconexão de componentes.

O controle do fluxo de dados que passa por uma porta de comunicação é realizado pela própria porta, simplificando os componentes interconectados. De forma a gerenciar os fluxos de dados, as portas de comunicação suportam duas semânticas para o tratamento dos dados: mensagens e fluxo contínuo.

A Figura 10 ilustra a semântica de orientação a mensagens. O componente transmissor envia os dados através da porta de saída, que por sua vez os encaminha, via canal de comunicação, para a porta de entrada correspondente. Na abordagem orientada a mensagens, as portas de comunicação organizam os dados em uma seqüência de mensagens individuais e o componente receptor recupera estas mensagens, uma de cada vez, na ordem em que elas foram enviadas pelo componente transmissor.

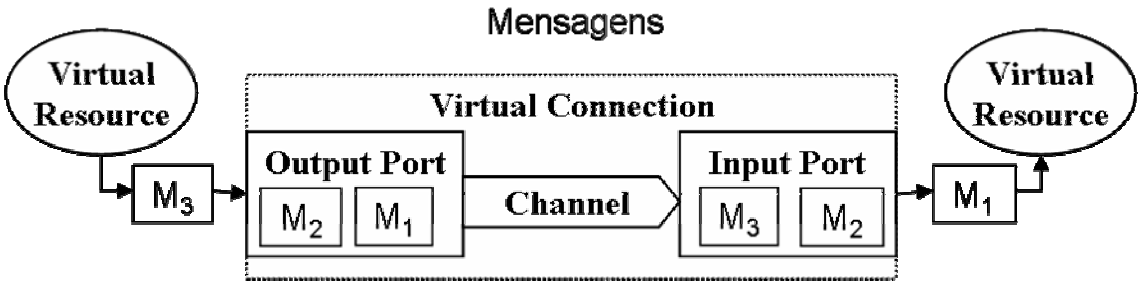

Figura 10. Orientação a Mensagens.

Já na abordagem orientada a fluxo, apresentada na Figura 11, as portas organizam os dados como uma seqüência de octetos individuais, mantendo a ordem de envio destes octetos. A porta de saída recebe uma série de octetos, e da mesma maneira que na orientação a mensagem, transmite estes octetos para a porta de entrada através do canal de comunicação.

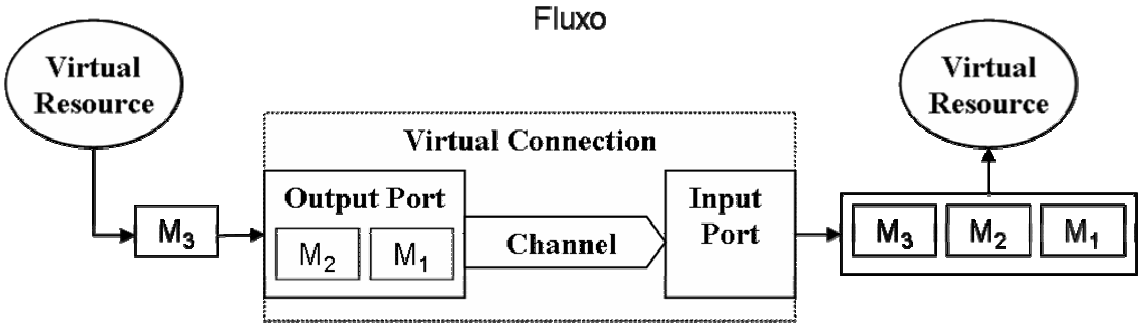

Figura 11. Orientação a Fluxo

A porta de entrada organiza estes octetos na mesma ordem de recebimento, e ao receber uma chamada do componente receptor, entrega todo o conteúdo de seus *buffers* de recepção com uma seqüência de octetos na mesma ordem de recebimento.

Ainda levando em consideração a interação entre componentes e portas, o modelo define também duas semânticas operacionais: síncrona e assíncrona. A semântica síncrona funciona de maneira bloqueante, onde o componente chamador fica bloqueado até que a porta de entrada receba algum dado ou que a porta de saída consiga tratar os dados que foram enviados pelo componente. Por outro lado, na semântica assíncrona, o componente chamador não fica bloqueado ao tentar ler de uma porta de entrada que não contém dados, ou ao tentar enviar algo por uma porta de saída que não tenha espaço disponível para armazenar os dados até que esses possam ser efetivamente enviados.

O modelo de interconexão provê um mecanismo de suporte à adaptação para o *framework* Cosmos, suportando os dois tipos de adaptação definidos pelo *framework*: reativa e pró-ativa. O objetivo é permitir a reconfiguração dinâmica dos componentes envolvidos mantendo a sincronização entre os fluxos, de forma que o fluxo não seja interrompido. Para isso, utiliza-se a clonagem do canal de comunicação.

A Figura 12 apresenta um exemplo de uma interconexão do ponto de vista da conexão virtual, envolvendo uma porta de entrada e uma porta de saída. A conexão virtual mantém referências para as portas e os canais de comunicação envolvidos. Este cenário servirá de exemplo para o funcionamento de um processo de adaptação.

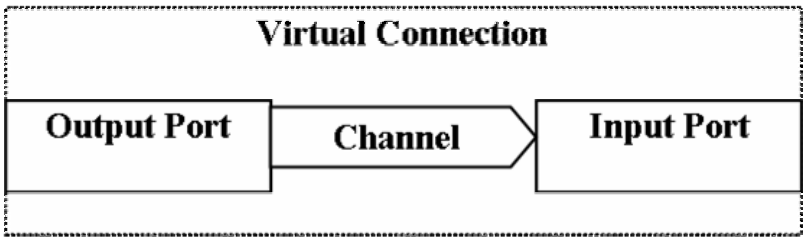

Figura 12. Conexão Virtual com um Canal de Comunicação.

Uma adaptação é controlada pelo *Configurator*, que verifica a consistência da nova configuração antes de efetivá-la. Após esta verificação, o *Configurator*, através da conexão virtual, clona os canais de comunicação existentes. O objetivo desta clonagem é não interromper o fluxo de mídia enquanto os componentes estão alterando seus parâmetros. Com os canais de comunicação clonados, a conexão virtual notifica às

portas que passam a operar com dois canais de comunicação. A Figura 13 apresenta o estado da conexão virtual após a clonagem do canal de comunicação.

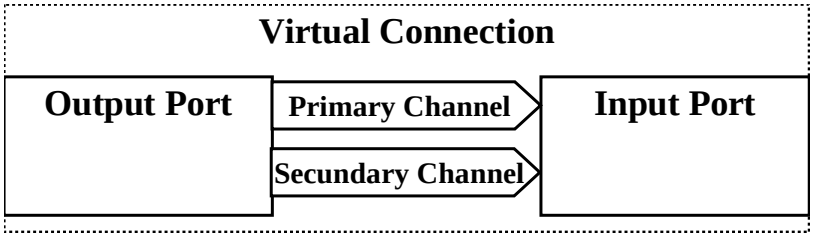

Figura 13. Conexão Virtual com o canal de comunicação clonado.

O canal primário (*Primary Channel*) corresponde ao canal de comunicação antigo, por onde é transmitido o fluxo original, enquanto que o canal secundário (*Secundary Channel*) corresponde ao canal de comunicação recém-criado através da clonagem do canal primário. Este canal secundário é utilizado para a transmissão do novo fluxo de mídia, enquanto ainda houver dados no canal primário.

Os componentes, ao serem notificados pelo *Configurator*, iniciam os preparativos para a troca de seus parâmetros, enquanto continuam operando com o fluxo de mídia antigo no canal primário. Após esta fase de alocação, o *Configurator* identifica o componente transmissor, notificando o mesmo para que ele troque seus parâmetros. Neste momento, o componente transmissor pára a transmissão e troca seus parâmetros operacionais, para, em seguida, notificar a sua respectiva porta de saída que os parâmetros foram trocados e reinicia a transmissão.

A porta de saída, ao receber a notificação do componente transmissor, realiza a troca do canal de comunicação ativo, passando a enviar dados pelo canal de comunicação secundário; em seguida, descarta o canal primário, que poderá eventualmente conter dados em trânsito. Neste momento, a porta de entrada estará recebendo dados por dois canais distintos, armazenando estes dados em *buffers* de recepção diferentes. O componente receptor continua recebendo dados de sua porta de entrada normalmente, até que o canal de comunicação primário não entregue mais dados, e o buffer de recepção primário da porta de entrada se esvazie.

Neste momento, a porta de entrada notifica ao seu componente receptor, e realiza a troca do canal de comunicação ativo, descartando o canal primário. Com esta notificação, o componente receptor realiza a troca de seus parâmetros operacionais, e retorna a sua operação normal, solicitando dados de sua porta de entrada.

Com a notificação de suas duas portas de comunicação, o canal notifica à conexão virtual, que, por sua vez, desaloca o canal de comunicação primário. O canal secundário passa a operar então como canal primário, e os componentes passam a operar normalmente. Esta configuração final, com o canal primário descartado e o canal secundário passando a ser o único canal existente é exibida na Figura 14.

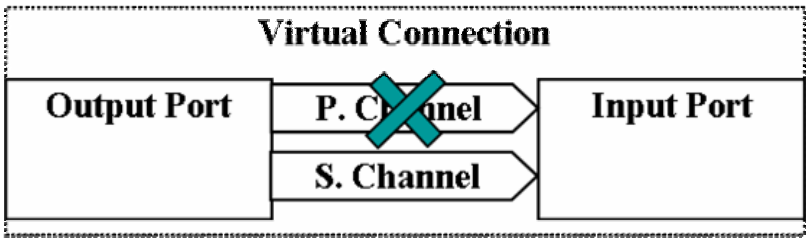

Figura 14. Conexão Virtual com o Canal de Comunicação Secundário ativo.

3.2. Componentes do Modelo de Interconexão

Esta seção apresenta os componentes definidos pelo modelo de interconexão e o relacionamento entre eles através da especificação de suas interfaces. A definição dos componentes considera os requisitos de suporte à adaptação dinâmica do *framework* Cosmos. Esta adaptação pode acontecer de duas maneiras: através da reconfiguração interna de um componente com a troca de seus parâmetros operacionais, ou através da substituição de componentes.

A definição do modelo apresentado focou a adaptação interna de um componente através da troca de seus parâmetros operacionais. Desta forma, a definição das interfaces dos componentes do modelo leva em consideração este tipo de adaptação. Para a definição dos nomes de interfaces foi adotada a padronização de se utilizar o prefixo *I* seguido do nome da interface ou do componente.

Primeiramente, o modelo é apresentado sob um ponto de vista estrutural, de forma a ilustrar os componentes envolvidos e suas dependências. Em seguida, é feita uma explanação do ponto de vista comportamental, ilustrando as interfaces providas e requeridas por cada componente do modelo.

A Figura 15 apresenta uma visão estrutural do modelo de interconexão através de um diagrama de componentes simplificado. Essa visão estrutural explicita o arranjo topológico entre os seus elementos constituintes seguindo as regras definidas pelo *framework* Cosmos. Todos os componentes do *framework* devem oferecer a interface

IBasicService, que é a interface de serviços básicos do *framework*, sendo utilizada para adquirir as interfaces funcionais dos componentes [4]. Este diagrama oferece uma visão geral da estrutura do modelo explicitando o caminho que o fluxo percorre de um componente para o outro.

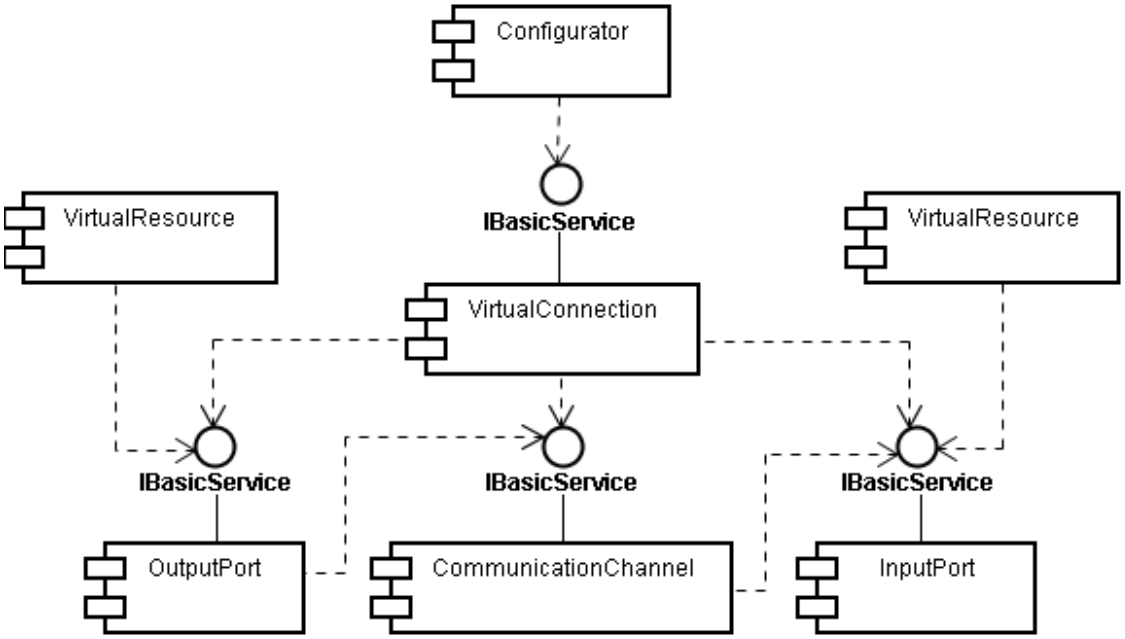

Figura 15. Visão Estrutural do Modelo de Interconexão

O *framework* Cosmos oferece uma arquitetura conceitual para o projeto e o desenvolvimento de plataformas multimídia distribuídas, com suporte à configuração e adaptação dinâmica através do uso de reflexividade. A adaptação permite a mudança nos parâmetros de funcionamento dos componentes, tais como a mudança do formato dos fluxos em tempo de execução. Para garantir o suporte a este tipo de reconfiguração, o modelo utiliza a técnica de criação de novos canais de comunicação, que são utilizados para o envio dos dados no novo formato. Esta técnica foi definida na proposta para facilitar o gerenciamento da sincronização do fluxo de dados durante a mudança de formato dos fluxos.

A Figura 16 apresenta uma visão geral do modelo de interconexão sob um ponto de vista comportamental, exibindo as principais interfaces dos componentes envolvidos. O configurador, representado pelo componente *Configurator*, tem um papel crucial no modelo. Ele tem a responsabilidade de realizar a negociação entre os componentes

envolvidos de forma a cumprir as exigências da aplicação. Para tal, ele precisa considerar as potencialidades e limitações da plataforma.

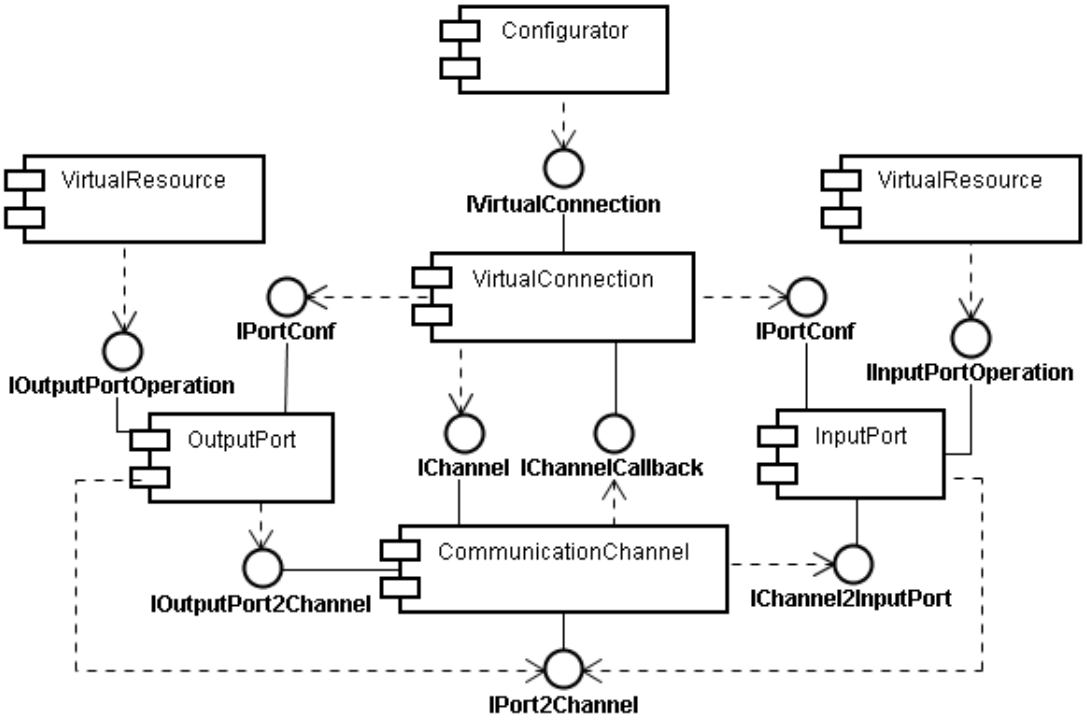

Figura 16. Visão Funcional do Modelo de Interconexão.

O configurador utiliza a interface *IVirtualConnection* para interagir com a conexão virtual, representada pelo componente *VirtualConnection*. A conexão virtual, por sua vez, realiza operações de gerenciamento nas portas de comunicação e nos canais, utilizando para isso, as interfaces *IPortConf* e *IChannel* respectivamente. Estas interfaces são utilizadas para gerenciar e configurar, respectivamente, as portas de comunicação e os canais envolvidos, sendo utilizadas também para realizar o processo de adaptação.

As interfaces *IOutputPortOperation* e *IInputPortOperation* são as interfaces operacionais oferecidas pelas portas de comunicação e utilizadas pelos recursos virtuais para enviar e receber dados respectivamente. Estas interfaces oferecem métodos para o envio e a recepção de dados com semânticas bloqueante e não-bloqueante, e para o funcionamento orientado a mensagens e a fluxo.

O canal de comunicação, representado pelo componente *CommunicationChannel*, realiza a ligação entre duas portas. Ele é utilizado pelas portas

para a troca de dados, abstraindo os detalhes referentes à tecnologia de comunicação suportada pela plataforma, como por exemplo, memória compartilhada para o caso de portas locais, e *sockets* UDP para portas remotas. A interface *IOutputPort2Channel* é utilizada pela porta de saída para enviar dados através do canal de comunicação. Por outro lado, a interface *IChannel2InputPort* é utilizada pelo canal para entregar os dados para a porta de entrada. O canal de comunicação utiliza a interface *IChannelCallback* para se comunicar com a conexão virtual, indicando que o canal não é mais necessário e que já pode ser desalocado.

A interface *IPort2Channel* oferecida pelo canal de comunicação é utilizada pelas portas para sinalizar ao canal que uma adaptação foi concluída, indicando que a porta sinalizadora não mais precisa daquele canal. Exemplos desta situação são a troca de canais devido a um processo de adaptação e a possível saída de um usuário de uma interconexão *multicast.*

O número de canais em uma conexão virtual depende da topologia de interconexão estabelecida pela especificação da aplicação. O modelo suporta conexões *unicast*, com relacionamento 1 para 1 entre a conexão virtual e o canal de comunicação, e conexões *multicast*, com relacionamento 1 para vários entre a conexão virtual e o canal de comunicação. A Figura 17 apresenta um exemplo envolvendo uma conexão multicast com um componente transmissor e dois componentes receptores. Como pode-se perceber, o canal de comunicação realiza a ligação entre cada par de portas de comunicação existentes, com uma porta podendo estar associada a diferentes canais.

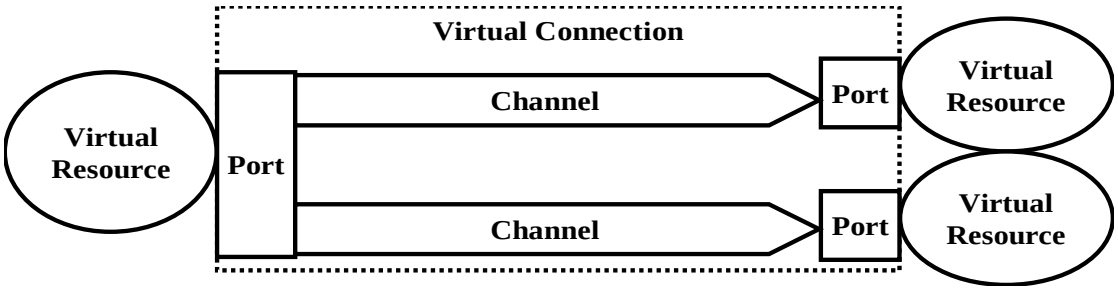

Figura 17. Exemplo de Conexão Multicast.

Para o estabelecimento de conexões *multicast*, o sistema aloca diversos canais de comunicação, onde cada canal pode adotar uma tecnologia de comunicação diferente. A escolha de uma tecnologia específica é tratada pelo *middleware* que implementa o *framework*. O *middleware* pode considerar, por exemplo, os tipos de formatos dos dados trocados na comunicação, requisitos temporais de mídia e sincronização e a

localização dos componentes participantes. Por exemplo, podem existir interconexões que estão no mesmo espaço de endereçamento, ou em espaços de endereçamento diferentes, mas em uma mesma máquina física ou em espaços completamente distribuídos.

O canal de comunicação realiza a ligação entre duas portas, independentemente da localização destas portas. Para isso, o componente canal é formado através de uma composição envolvendo três componentes: um gerente, denominando *ChannelManager*, e dois subcanais, que correspondem às pontas do canal de comunicação. Os subcanais são os componentes que efetivamente realizam a comunicação entre as portas.

A Figura 18 apresenta estes componentes e o seu relacionamento. A interface *IOutputPort2Channel*, utilizada pela porta de saída para envio de dados, é oferecida pelo componente *SourceSubChannel* e a interface *IChannel2InputPort*, utilizada pelo canal para a comunicação com a porta de entrada, é acessada pelo componente *TargetSubChannel*. Além disso, a interface *IPort2Channel*, utilizada pelas portas, é oferecida por ambos componentes *SourceSubChannel* e *TargetSubChannel*.

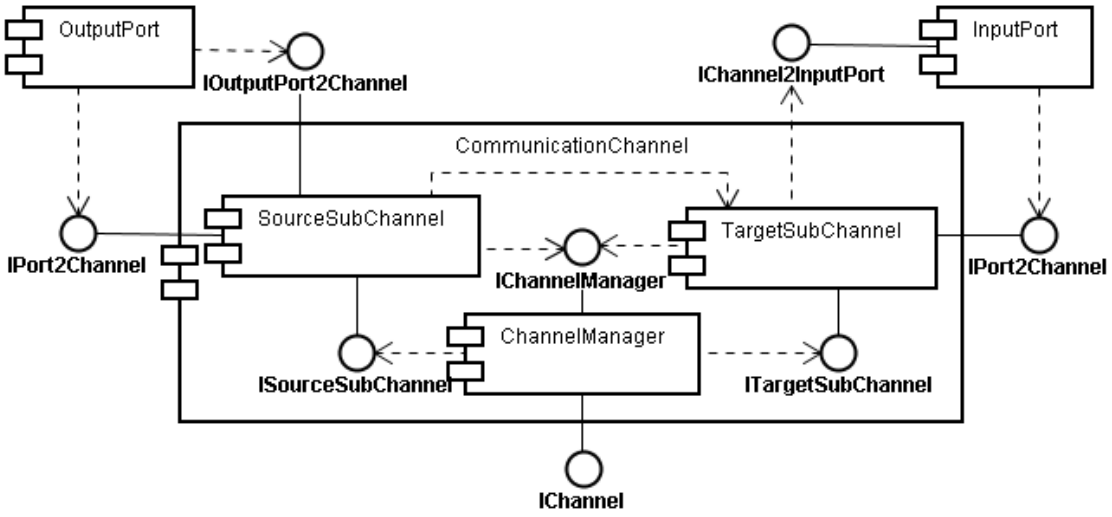

Figura 18. Componentes Internos de um Canal de Comunicação.

Uma porta de saída se comunica com o componente *SourceSubChannel*, enquanto que uma porta de entrada se comunica com o *TargetSubChannel*. Os subcanais residem no mesmo espaço de endereçamento que suas portas associadas.

O componente *ChannelManager* é o responsável pelas operações de gerenciamento de canal, tais como abertura e fechamento de um canal, servindo como

ponto de acesso para o canal de comunicação através da interface *IChannel* utilizada pela conexão virtual. Para isso, ele acessa os subcanais através das interfaces *ISourceSubChannel* para o componente *SourceSubChannel*, e *ITargetSubChannel* para o componente *TargetSubChannel*, enquanto que os subcanais utilizam a interface *IChannelManager* para se comunicarem com o *ChannelManager*. Com isso, as operações oferecidas pela interface *IChannel* podem ser delegadas para os subcanais quando necessário, e as operações da interface *IPort2Channel* também ativam o *ChannelManager* quando necessário, como por exemplo devido a saída de um componente de uma conexão *multicast*, ou durante a realização de uma adaptação. A comunicação entre os dois subcanais é realiza através dos mecanismos ou protocolo de comunicação que o canal representa.

3.3. Interfaces de interconexão

A arquitetura de interconexão proposta explora o conceito de propriedades definido pelo *framework* Cosmos para descrever requisitos e características de componentes e de portas, sendo usadas para fins de gerenciamento de configuração. Ela define um conjunto de interfaces padrão que se caracterizam como interface de configuração e interface de operação.

As interfaces de configuração são utilizadas pelo *middleware* que implementa o *framework* para operações de configuração e gerenciamento do ciclo de vida dos elementos envolvidos. As interfaces de operação são as interfaces disponibilizadas para os componentes clientes do *middleware*, pelas quais estes componentes se comunicarão com o ambiente ou com outros componentes, abstraindo detalhes referentes ao processo de interconexão e de transmissão de dados.

O comportamento operacional de uma porta de comunicação varia dependendo do seu tipo ser de entrada ou saída. Devido a isso, foram definidas duas interfaces operacionais para as portas de comunicação, *IOutputPortOperation* e *IInputPortOperation*. Estas interfaces operacionais permitem uma abstração acerca dos detalhes da tecnologia de comunicação e da topologia de interconexão.

A Figura 19 apresenta estas interfaces. A interface *IInputPortOperation* oferece uma API bem definida para a operação de uma porta de entrada, representando um ponto de acesso único para os componentes que necessitam deste tipo de porta. Esta interface operacional aborda questões relacionadas com a sincronização, oferecendo

operações bloqueantes e não-bloqueantes para a manipulação de fluxos e troca de mensagens. Conforme exibido na Figura 19, os métodos *getMessage* e *waitMessage* dão suporte à troca de mensagens, onde o componente receptor recupera exatamente uma mensagem por vez, na ordem em que foram enviadas pelo componente transmissor. De forma análoga, os métodos *getBuffer* e *waitBuffer* permitem a manipulação de fluxos, onde os componentes recuperam todo o conteúdo dos *buffers* de entrada. As operações com o prefixo *get* (*getMessage* e *getBuffer*) são as chamadas que funcionam de maneira não-bloqueante, enquanto que as operações com o prefixo *wait* (*waitMessage* e *waitBuffer*) possuem uma semântica bloqueante.

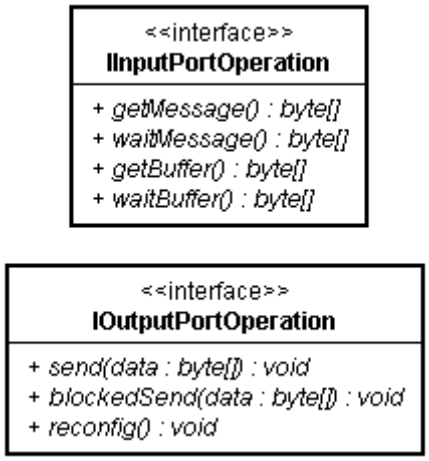

Figura 19. Interfaces de Operação das Portas de Comunicação.

A interface *IOutputPortOperation*, de maneira análoga a *IInputPortOperation*, oferece uma API bem definida para a operação de uma porta de saída, representando um ponto de acesso unificado para os componentes que possuem este tipo de porta. Sua API é apresentada na Figura 19, onde se encontram métodos para o envio de dados, de forma bloqueante (*blockedSend*) e não-bloqueante (*send*), e o método *reconfig*, que é utilizado pelo componente emissor no momento da adaptação de forma a notificar a porta que já pode realizar a troca do canal.

O conceito de propriedades é suportado através da definição de um modelo de meta-dados que é utilizado pelos componentes do *framework* para a representação dos componentes concretos. Este modelo é utilizado pelo *ApplicationProxy* para o armazenamento do estado atual da aplicação conforme definido pelo Cosmos.

Durante a interpretação da especificação da aplicação, as informações referentes às conexões definidas são utilizadas para preencher os meta-dados do componente

VirtualConnection. Uma conexão virtual permite a adição de uma nova porta depois de iniciada a sua execução, se assim prever a descrição da conexão na especificação da aplicação, podendo possuir várias portas de saída e entrada devido às diversas topologias de interconexão suportadas.

A Figura 20 apresenta a API definida para o componente *VirtualConnection*, correspondendo às operações da interface *IVirtualConnection*. Esta interface define métodos que são utilizados para o gerenciamento da conexão. Ela define métodos para adicionar, listar, recuperar ou remover portas de uma conexão. Estes métodos de gerenciamento podem ser utilizados, por exemplo, para a entrada ou saída de um componente em uma conexão *multicast*, como em uma aplicação de vídeo-aula.

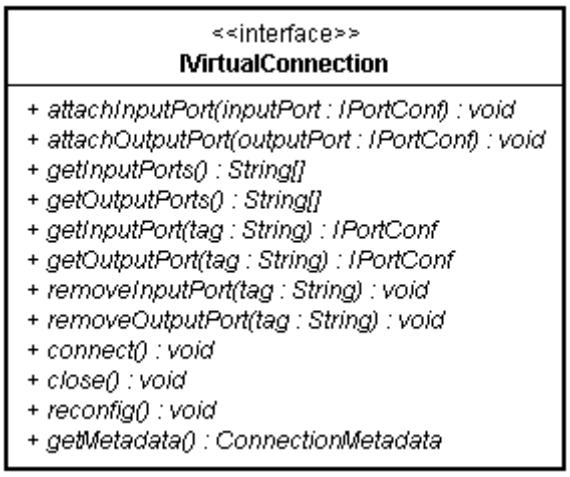

Figura 20. Interface *IVirtualConnection.*

A operação *attachInputPort* recebe uma referência para uma porta de entrada através da interface *IPortConf*, enquanto que o método *attachOutputPort* recebe uma referência para uma porta de saída. Estes métodos armazenam estas referências em listas específicas para cada tipo de porta, entrada ou saída. Os métodos *getInputPorts* e *getOutputPorts* são utilizados para retornar a lista de portas de entrada e saída, respectivamente. Estas operações retornam uma lista com os identificadores das portas, e não as referências das portas em si. Para isso, é utilizado o modelo de meta-dados definido pelo *framework* Cosmos, explorando o conceito de propriedades.

O método *getInputPort* é utilizado para recuperar a referência de uma porta de entrada, enquanto que *getOutputPort* retorna uma referência de uma porta de saída. Estes métodos recebem como parâmetro o identificador da porta desejada (*tag*), e

retornam uma referência da porta correspondente utilizando a interface *IPortConf*. De maneira análoga, os métodos utilizados para remover uma porta da conexão recebem como parâmetro o identificador da porta através da string *tag*. O método *removeInputPort* remove a porta de entrada indicada pelo parâmetro da conexão, enquanto que o método *remoteOutputPort* remove uma porta de saída.

O método *connect* é utilizado para iniciar o processo de interconexão, onde a conexão virtual aloca os recursos necessários para a instanciação dos canais de comunicação e configura os canais criados com suas respectivas portas de comunicação. O método *close* é utilizado para finalizar uma interconexão, liberando neste caso os recursos alocados, e fechando os canais de comunicação. O método *reconfig* é utilizado pelo *Configurator* para dar início a um processo de adaptação, onde os canais de comunicação serão duplicados e as portas reconfiguradas para operar simultaneamente com dois canais de comunicação durante o processo de adaptação. A operação *getMetadata* é utilizada para recuperar os meta-dados associados a uma conexão virtual. Seu valor de retorno (*ConnectionMetadata*) corresponde ao elemento que representa os meta-dados de uma conexão virtual, conforme definido pelo *framework* Cosmos.

O componente *VirtualConnection* usa a interface *IPortConf*, apresentada na Figura 21, para configurar as associações entre os canais de comunicação e seus respectivos pares de portas. Esta interface é utilizada também durante o processo de adaptação dinâmica, onde as portas de comunicação precisam realizar alguns procedimentos de forma a se prepararem para este processo. Ela define métodos para o gerenciamento dos canais de comunicação associados à porta, permitindo que se adicione um canal de comunicação, que se recupere a lista de canais associados a esta porta, bem como a recuperação de um dos canais associados e sua remoção.

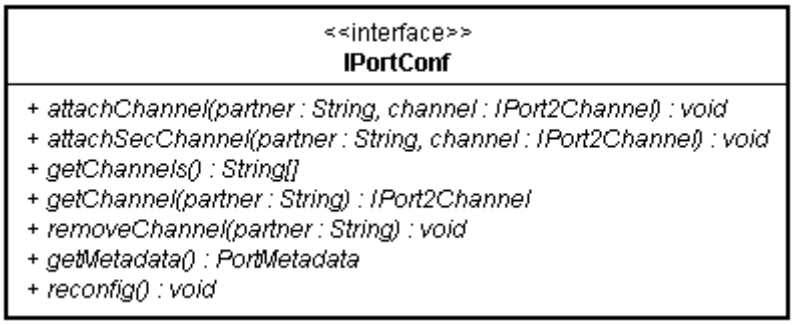

Figura 21. Interface *IPortConf*.

O método *attachChannel* é utilizado durante o estabelecimento da conexão para indicar à porta o seu respectivo canal. Esta operação recebe como parâmetro um identificador (*partner*) que representa a porta de comunicação associada. Este identificador é utilizado para diferenciar os possíveis canais de comunicação junto à porta. A operação *attachSecChannel* coloca a porta em estado de adaptação, e é utilizada para associar o canal secundário à porta. Neste momento, a porta de comunicação passa a trabalhar com dois canais simultaneamente, diferenciando no tratamento adotado aos dados transportados. Caso seja uma porta de saída, os canais são trocados após a notificação por parte do componente transmissor para a porta. Uma porta de entrada passa a trabalhar com dois *buffers* distintos até que o canal primário possa ser desalocado.

O método *getChannels* retorna a lista de canais de comunicação associado à porta em questão. Este método retorna a lista de identificadores associados aos canais. O método *getChannel* recebe como parâmetro um identificador (*partner*) e retorna a referência do canal correspondente utilizando a interface *IPort2Channel*. A operação *removeChannel* é utilizado para remover um canal da porta de comunicação e, de maneira semelhante à *getChannel*, recebe como parâmetro um identificador que indica o canal a ser removido.

O método *getMetadata* é utilizado para recuperar os meta-dados associados à porta de comunicação. De maneira análoga ao método *getMetadada* da interface *IVirtualConnection*, o valor de retorno corresponde ao elemento que representa os meta-dados associados às portas de comunicação seguindo a hierarquia de meta-dados definida pelo *framework* Cosmos. A operação *reconfig* é utilizada pelo componente cliente da porta de comunicação para notificar a porta que os parâmetros operacionais do componente foram alterados e que a porta pode realizar a troca dos canais de comunicação.

A interface *IChannel*, apresentada na Figura 22, é utilizada pelo componente *VirtualConnection* para operações de configuração e gerenciamento dos canais de comunicação. Ela é oferecida pelo componente *CommunicationChannel*, provendo operações que permitem o gerenciamento de um canal de comunicação. A operação *attachTarget* indica ao canal sua porta de entrada associada. Este método recebe como parâmetro o identificador da porta de comunicação, e é delegado para o subcanal associado à porta de entrada (*TargetSubChannel*). Como os subcanais se encontram no mesmo espaço de endereçamento que suas respectivas portas, eles se reportam ao

Configurator da plataforma, utilizando o identificador recebido como parâmetro, de forma a adquirir a referência da porta de comunicação associada.

O método *getTarget* retorna um identificador para a porta de entrada associada ao canal. Como esta interface é utilizada pela conexão virtual, que possui referências para as portas associadas, este método não retorna a referência da porta em questão. O método *open* é utilizado para abrir um canal de comunicação, onde operações específicas da tecnologia de comunicação que o canal representa são executadas. Por exemplo, em um canal UDP esta operação indica ao canal que o *socket* UDP já pode ser iniciado. De maneira análoga, o método *close* também é dependente da tecnologia de comunicação que o canal representa, sendo utilizado para o encerramento de uma conexão, indicando que todos os recursos utilizados para comunicação já podem ser liberados.

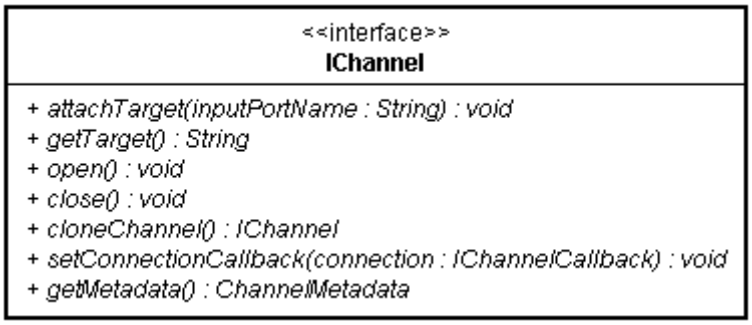

Figura 22. Interface *IChannel*.

O método *cloneChannel* é utilizado durante o processo de adaptação para duplicar o canal de comunicação, permitindo a criação de um canal secundário com as mesmas características do canal anterior, que será utilizado para a transmissão do novo fluxo de dados durante o processo. A operação *getMetadata* permite e recuperação dos meta-dados associados ao componente canal, seguindo a hierarquia definida pelo Cosmos.

Ao término do processo de adaptação, o canal secundário passa a ser o canal primário, e o canal antigo é descartado. A operação *setConnectionCallback* é utilizada para indicar ao canal de comunicação sua conexão virtual, de forma que o canal possa notificar a conexão virtual que não é mais necessário, e que já pode ser desalocado. A interface *IChannelCallback*, apresentada na Figura 23, é utilizada pelo canal de

comunicação para realizar esta notificação, oferecendo a operação *callback* que recebe como parâmetro o identificador do canal em questão.

Figura 23. Interface IChannelCallback.

As portas se comunicam com os canais de comunicação para o envio e recepção de dados, além de notificarem o canal acerca de sua liberação. A Figura 24 apresenta as interfaces utilizadas na comunicação entre as portas e os canais.

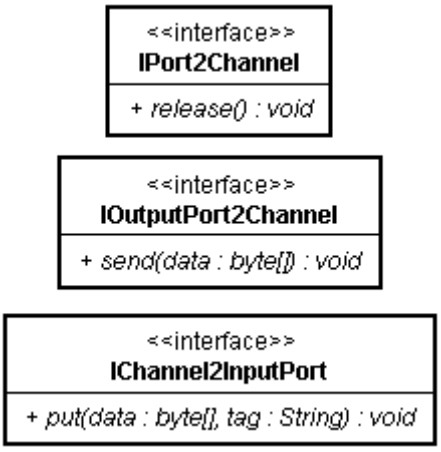

Figura 24. Interfaces para Comunicação entre as Portas e o Canal de Comunicação.

A interface *IPort2Channel* é utilizada para a notificação ao canal de sua liberação, informando que ele pode ser desalocado. Isto é utilizado durante o processo de adaptação, quando a porta de comunicação realiza a troca dos canais ativos, e passa a operar com o canal secundário. O canal de comunicação aguarda esta notificação de suas duas portas associadas, de forma a garantir que o canal não está mais sendo utilizado pelas mesmas.

A interface *IOutputPort2Channel* é utilizada pela porta de saída para o envio de dados para o canal de comunicação. Este método é implementado pelo subcanal correspondente à porta de saída. A interface *IChannel2InputPort* é oferecida pela porta de entrada e acessada pelo canal para a entrega de dados. De maneira análoga, o

subcanal correspondente às portas de entrada é quem acessa esta interface entregando os dados recebidos através da tecnologia de comunicação utilizada pelo canal.

O canal de comunicação é formado através da composição de um gerente e dois subcanais. Os subcanais representam as extremidades do canal de comunicação, residindo no mesmo espaço de endereçamento que suas portas associadas. As portas se comunicam com esses subcanais, os quais ficam sob a responsabilidade do canal. A Figura 25 apresenta as interfaces utilizadas para a comunicação entre o *ChannelManager* e seus subcanais. A interface *ISourceSubChannel* é utilizada pelo *ChannelManager* para se comunicar com o subcanal associado à porta de saída (*SourceSubChannel*). Ela oferece operações para a abertura (*open*) e o encerramento (*close*) deste subcanal. A interface *ITargetSubChannel* é oferecida pelo subcanal associado à porta de entrada (*TargeSubChannel*), e é utilizada pelo *ChannelManager* de maneira análoga à interface *ISourceSubChannel*. A operação *attachTarget* tem o papel de indicar ao subcanal sua porta de entrada associada. Esta operação é delegada pelo *ChannelManager* para o subcanal, que por sua vez interage com o *Configurator* responsável pelo sua plataforma de execução para recuperar a referência da porta de entrada indicada.

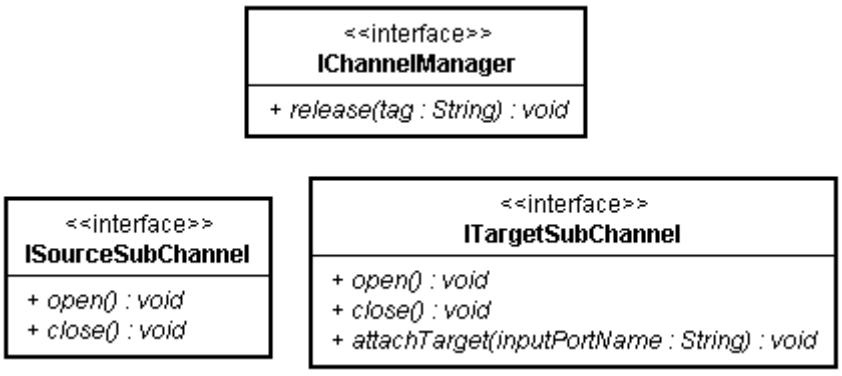

Figura 25. Interfaces para Comunicação entre o *ChannelManager* e os Subcanais.

As portas de comunicação utilizam a interface *IPort2Channel* para notificar o canal de comunicação que o mesmo não está mais sendo utilizado. Os subcanais, ao receberem esta notificação, utilizam a interface *IChannelManager* para notificar o componente *ChannelManager*. O método *release* recebe como parâmetro a identificação do subcanal, de forma que o canal saiba o momento em que seus dois subcanais não estão mais sendo utilizados, e possa notificar a conexão virtual que o canal de comunicação está liberado para ser desalocado.

3.4. Modelo de Interação dos Componentes

Como mencionado anteriormente, o modelo de interconexão apresenta um alto nível de flexibilidade suportando a adaptação dinâmica no nível de conexão virtual. Ele permite a adição ou remoção de portas em uma conexão, bem como a configuração de parâmetros de QoS definidos pela especificação, utilizando para isso as operações de propriedades e o *framework* de gerenciamento de QoS apresentado em [31]. A presente seção discute os principais componentes definidos pelo modelo descrevendo as funções desempenhadas por cada componente.

A Figura 26 apresenta um diagrama introdutório com as principais ações relacionadas com o estabelecimento de uma conexão.

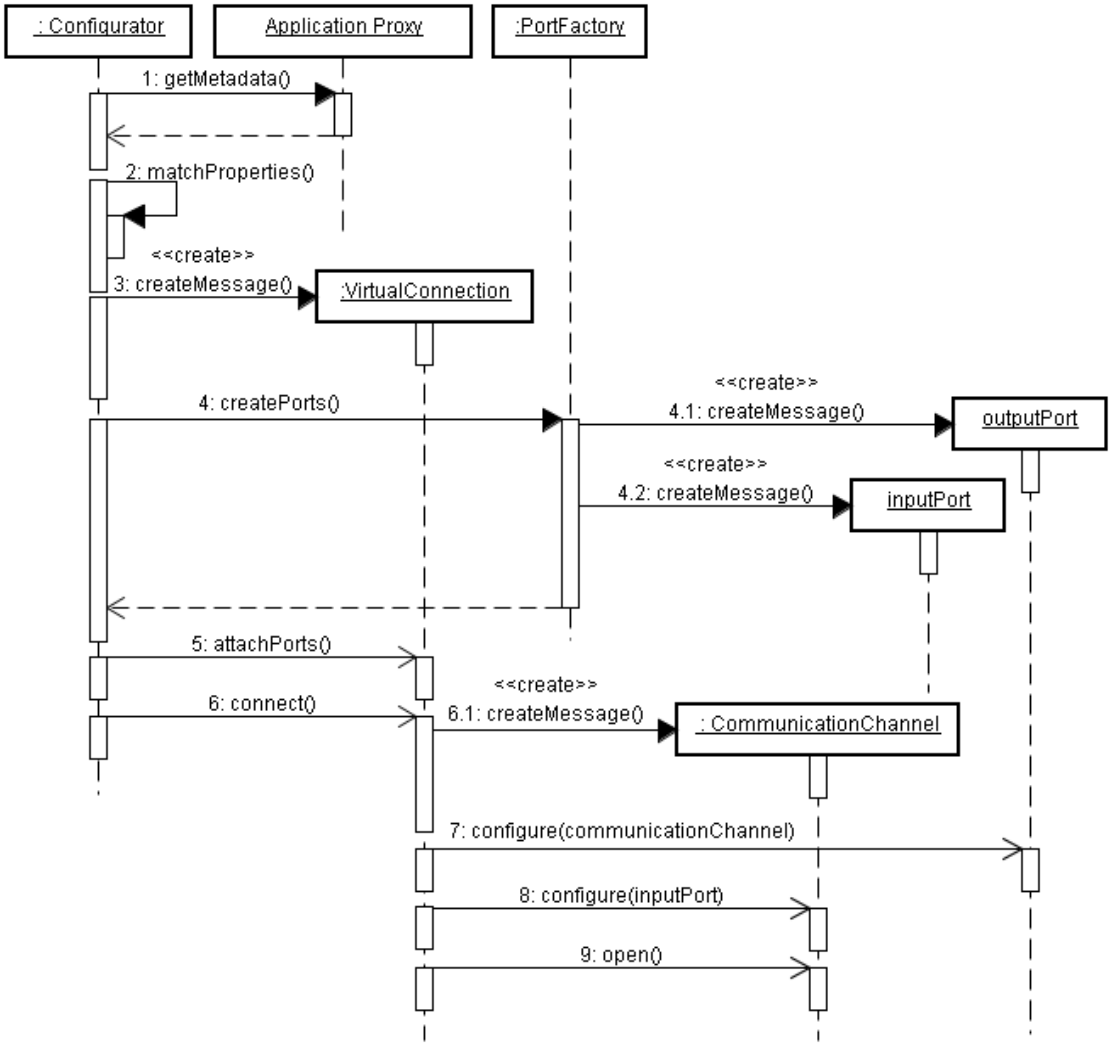

Figura 26. Processo Simplificado de Estabelecimento de uma Conexão.

A Figura 26 envolve ações de negociação de propriedades, instanciação de componentes e sua configuração. Seguindo o modelo definido pelo *framework* Cosmos, os meta-dados com as informações de configuração de uma aplicação, relacionadas com suas interconexões e propriedades das portas definidas, são obtidos através do processamento de uma especificação por um *parser* e mantidos durante o ciclo de vida da execução no *ApplicationProxy*. O *Configurator* recupera as informações referentes à conexão, representado pela operação *getMetadata*, e inicia o processo de negociação para a definição de acordos entre os componentes envolvidos, representado pela chamada *matchProperties*, para, em seguida, realizar a instanciação dos componentes.

De posse dos meta-dados referentes à conexão, o *Configurator* instancia o componente *VirtualConnection*, representado pela chamada *CreateMessage* para o elemento *VirtualConnection*, para em seguida instanciar as portas necessárias para o estabelecimento da conexão. A chamada *createPorts* representa o processo de criação das portas de comunicação através de um componente do tipo *factory* (*PortFactory*). Este processo esta representado pelas operações *createMessage* para os elementos *outputPort* e *inputPort*, e será mais detalhado nas próximas seções. Após a criação das portas de comunicação, elas são registradas junto ao componente *VirtualConnection*. Este registro está representado pela chamada *attachPorts*, onde a conexão virtual recebe portas de entrada e portas de saída através de chamadas distintas. Em seguida, o *Configurator* inicia o processo de interconexão junto ao *VirtualConnection* através da chamada *connect*. Esta chamada inicia o processo de alocação dos canais de comunicação que forem necessários, de acordo com os parâmetros definidos pelo *middleware* que implementa o *framework*. Para cada par de portas (entrada e saída), a conexão virtual instancia um canal de comunicação diferente. O processo de alocação de um canal de comunicação esta representado pela chamada *createMessage* para o elemento *CommunicationChannel*.

Após a criação do canal de comunicação, a porta de saída é configurada com seu canal associado, representado pela chamada *configure* para o elemento *outputPort* que tem como parâmetro o elemento *CommunicationChannel*. Em seguida, o mesmo é configurado com a porta de entrada que receberá os dados do canal, representado pela chamada *configure* para o elemento *CommunicationChannel*, recebendo como parâmetro, o elemento *inputPort*. Na seqüência, a conexão virtual realiza a chamada *open* para iniciar as operações do canal de comunicação.

O modelo suporta a comunicação entre componentes de maneira transparente para os mesmos, utilizando para isso as definições do *framework* Cosmos. O *configurator* é o elemento central do *framework*, tendo controle de todos os recursos da plataforma. O Cosmos prevê também a comunicação entre Configuradores, no caso de uma aplicação distribuída, de forma a abstrair os detalhes referentes à distribuição dos componentes envolvidos. Isto acontece porque o *Configurator* é o elemento responsável pelos recursos em seu espaço de endereçamento, autorizando ou não a instanciação de um novo componente, de acordo com os recursos disponíveis na plataforma. Para tal, é necessário que ele tenha um registro de todos os componentes instanciados em seu espaço de endereçamento, de forma a manter o controle dos recursos utilizados.

O modelo utiliza esta comunicação entre configuradores para oferecer suas funcionalidades de maneira transparente no que diz respeito à distribuição. Assim, o comportamento dos componentes envolvidos em uma interconexão deve ser o mesmo, independentemente de a comunicação ser local ou distribuída. Para isso, foram analisados diversos cenários de instanciação dos componentes do modelo de interconexão, de forma a garantir a sua transparência.

O texto a seguir apresenta os possíveis cenários considerando os tipos de componentes do modelo. Primeiramente, é apresentado um processo genérico de instanciação de um componente, onde são explicitados os passos do *Configurator*. Em seguida são apresentados os passos de criação para componentes específicos.

3.4.1. Criação de Componentes

O *Configurator* é o componente central, conforme definido pelo *framework* Cosmos; ele é o elemento responsável pelos recursos da plataforma, devendo, portanto, tomar conhecimento sobre quaisquer componentes instanciados dentro de seu espaço de endereçamento. Devido a isso, a instanciação de componentes deve passar pela avaliação do *Configurator*, que pode validar, ou não, a instanciação, de acordo com os recursos disponíveis na plataforma. Para tal, foi definido um modelo padrão para a criação de componentes, onde o *Configurator* é o elemento acionado pelo componente solicitante da criação.

A Figura 27 apresenta a seqüência de operações desempenhadas pelo *Configurator*, de forma simplificada, durante o processo de instanciação de um componente. O *ApplicationProxy* é o componente responsável por manter os meta-dados da aplicação [4]. Após a negociação de propriedades e da verificação da

consistência da configuração da aplicação por parte do *Configurator*, o mesmo inicia o processo de criação de componentes. O *Configurator* recupera os meta-dados correspondentes ao componente cuja instanciação é solicitada (*getMetadata*) e inicia o processo de criação.

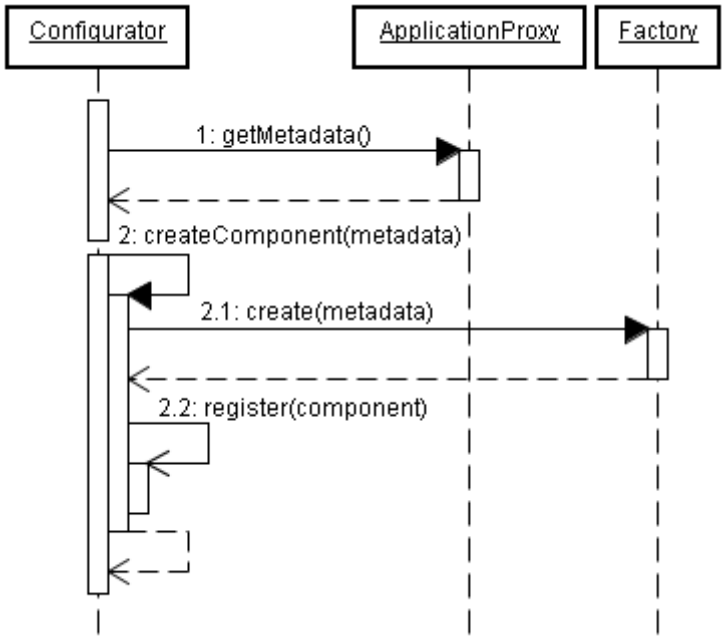

Figura 27. Processo Simplificado de Instanciação de Componentes.

O processo de criação é representado através da operação *createComponent*, recebendo como parâmetro os meta-dados recuperados do *ApplicationProxy*. De posse destes meta-dados, o *Configurator* identifica a fábrica adequada e instancia o componente correspondente. Após a instanciação do componente, o mesmo é registrado junto ao *Configurator*, de forma que se tenha um registro de todos os componentes instanciados em seu espaço de endereçamento. Este diagrama apresenta de maneira simplificada o processo genérico de criação de componentes no *framework* Cosmos; neste processo não é considerada a localização do componente, ou seja, o foco consiste apenas em mostrar os passos a serem seguidos pelo *Configurator* no momento de criação de um componente.

A Figura 28 apresenta o processo de criação de maneira detalhada, para o caso especifico de um componente local. A operação *createComponent* recebe como parâmetro os meta-dados recuperados do *ApplicationProxy* através da operação

getMetadata. Seu papel é identificar a localização do componente em questão através das propriedades em seus meta-dados.

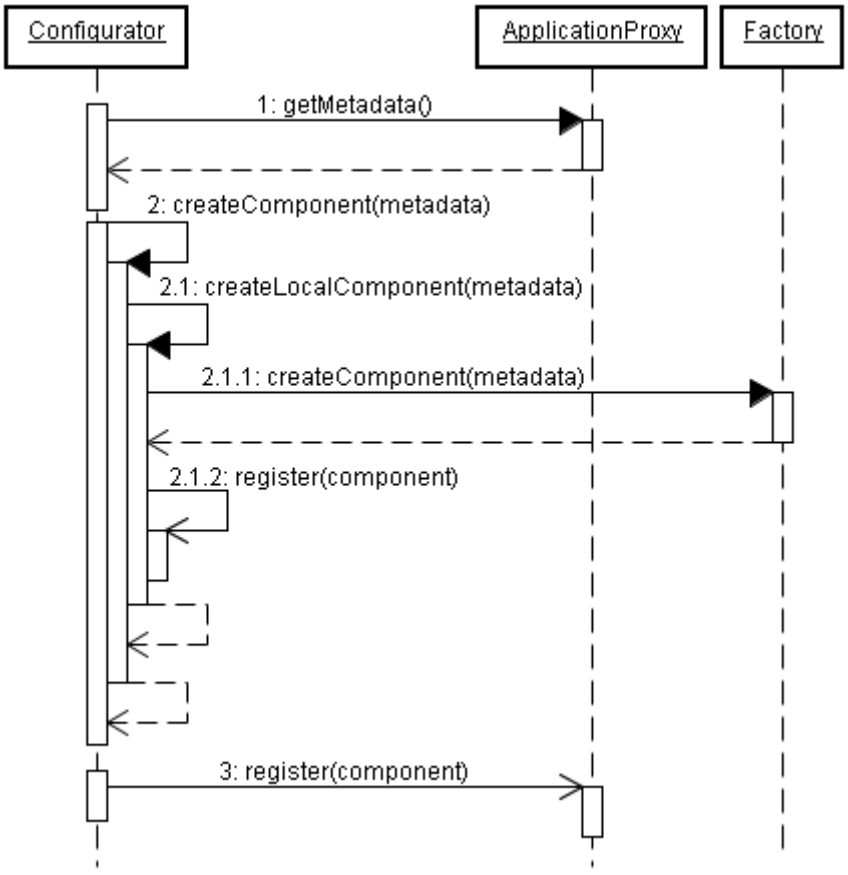

Figura 28. Processo de Criação de um componente Local.

Caso o componente se encontre no mesmo espaço de endereçamento do *Configurator*, o método *createLocalComponent* é chamado. Este método tem o papel de identificar o componente a ser criado e, utilizando sua fábrica correspondente, realiza a criação do mesmo.

O método *register* representa o registro do componente recém-criado na lista de componentes ativos do *Configurator*, de forma que ele tenha controle de todos os recursos e componentes instanciados em seu espaço de endereçamento. A última operação do diagrama é usada para registrar o componente recém-criado no *ApplicationProxy*. Como mencionado, o registro no *Configurator* serve para ele manter o conhecimento de todos os recursos disponíveis na plataforma; já o registro no *ApplicationProxy* funciona como ponto de acesso ao componente, no que diz respeito às funcionalidades da aplicação.

O diagrama da Figura 29 apresenta o processo geral de criação de um componente do *framework* em uma plataforma distribuída. Como mencionado, os meta-dados são recuperados pelo *Configurator* através da operação *getMetadata*.

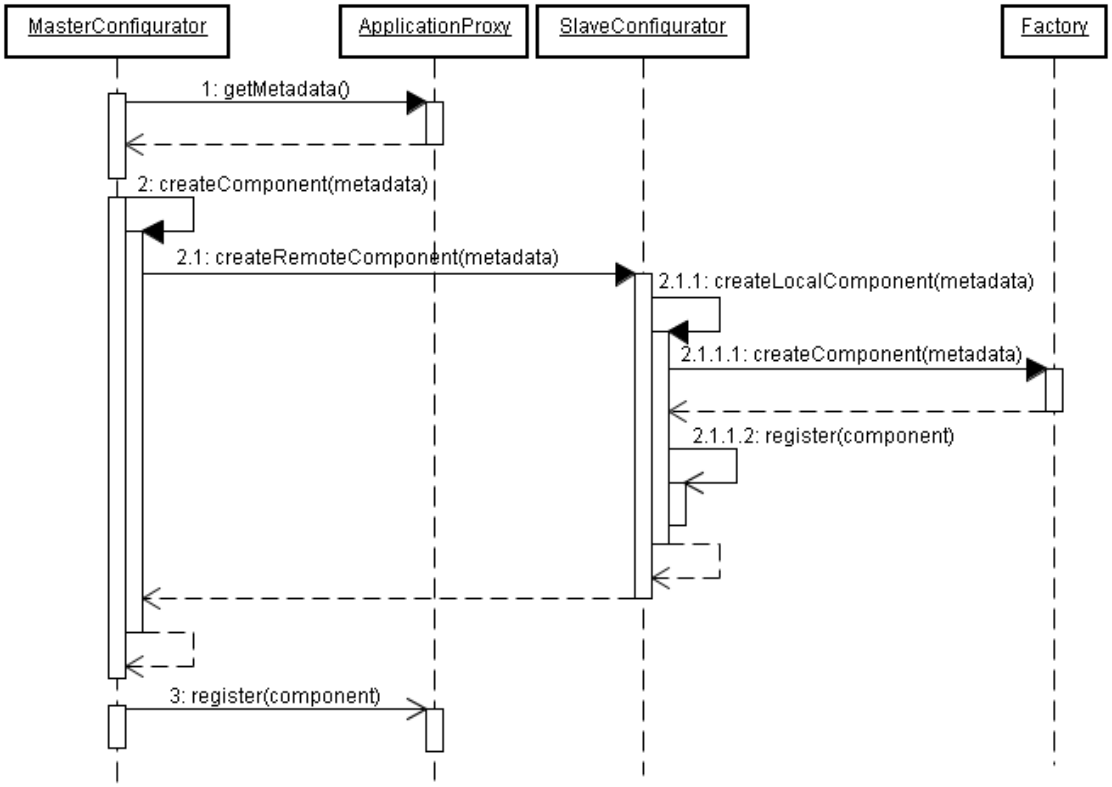

Figura 29. Processo de Criação de um Componente Distribuído.

A operação *createComponent* utiliza estes meta-dados, recebidos como parâmetro, para determinar a localização do componente e, no caso distribuído, realiza a comunicação com o *Configurator* remoto responsável pelos recursos do local em questão.

O *MasterConfigurator* representa o *Configurator* responsável pela aplicação, ou seja, todos os meta-dados referentes a esta aplicação estarão em um *ApplicationProxy* no domínio do *MasterConfigurator*. A operação *createComponent* abstrai a localização do componente a ser criado, devolvendo uma referência para uma interface oferecida pelo mesmo; é papel desta operação recuperar esta referência após a criação do componente, de forma que os detalhes referentes à distribuição dos componentes fiquem restritos à comunicação entre os configuradores do *middleware*. O *SlaveConfigurator* representa o *Configurator* do local onde o componente deve ser instanciado.

A chamada *createRemoteComponent* é uma chamada remota entre *Configurators* que deve seguir um protocolo de comunicação de acordo com o *middleware* que implementa o *framework* e com a tecnologia de distribuição utilizada. O *SlaveConfigurator* deve instanciar o componente em seu espaço de endereçamento, utilizando o processo já definido para um componente local apresentado na Figura 28. Neste caso, o *SlaveConfigurator* não precisa determinar a localização do componente em questão, realizando a chamada *createLocalComponent*, passando como parâmetro os meta-dados do componente. Como apresentado na Figura 28, a chamada *createLocalComponent* identifica a fábrica adequada e solicita a instanciação do componente, representada pela chamada *createComponent* para o elemento *Factory*.

Ao término do processo de instanciação de um componente local, o *Configurator* responsável pelo componente, *SlaveConfigurator*, deve registrá-lo localmente de forma a manter o controle de todos os recursos da plataforma local (operação *register* para o *SlaveConfigurator*), enquanto que o *MasterConfigurator* é responsável por adquirir a referência para a interface do componente remoto, representada pelo retorno da chamada *createRemoteComponent*, realizando o registro desta referência junto ao *ApplcationProxy* que se encontra em seu espaço de endereçamento (operação *register* para o *ApplicationProxy*).

3.4.2. Criação das Portas de Comunicação

O processo de instanciação de um componente no *framework* Cosmos segue os passos apresentados na Figura 28 para o caso de um componente local, e na Figura 29, para um componente remoto. A definição do componente a ser criado é realizada através dos meta-dados passados como parâmetros para as operações de criação, que identificam a fábrica adequada de acordo com o componente.

Por se tratar de um componente, o processo de instanciação de uma porta de comunicação segue os passos apresentados anteriormente, diferenciando no momento em que esses passos são executados pelo *Configurator*. A criação das portas de comunicação só é realizada após a negociação de propriedades, por parte do *Configurator*, e após a instanciação do componente *VirtualConnection* que gerencia as portas em questão.

A Figura 30 apresenta o processo de instanciação para o caso específico de uma porta de comunicação local, ou seja, no mesmo espaço de endereçamento que o *Configurator*. O *Configurator* recupera os meta-dados associados (*getMetadata*) e

realiza a chamada *createComponent*, passando como parâmetro estes meta-dados. Esta operação realiza uma verificação acerca da localização do componente a ser criado, no caso específico, a porta no mesmo espaço de endereçamento que o *Configurator*. Em seguida é realizada uma chamada para o método *createLocalComponent*, que tem o papel de identificar a fábrica adequada, de acordo com os meta-dados recebidos. No caso específico de uma porta, a fábrica ativada (operação *createPort*) é uma fábrica de portas (*PortFactory*), que instancia a porta de comunicação e retorna a referência para sua interface.

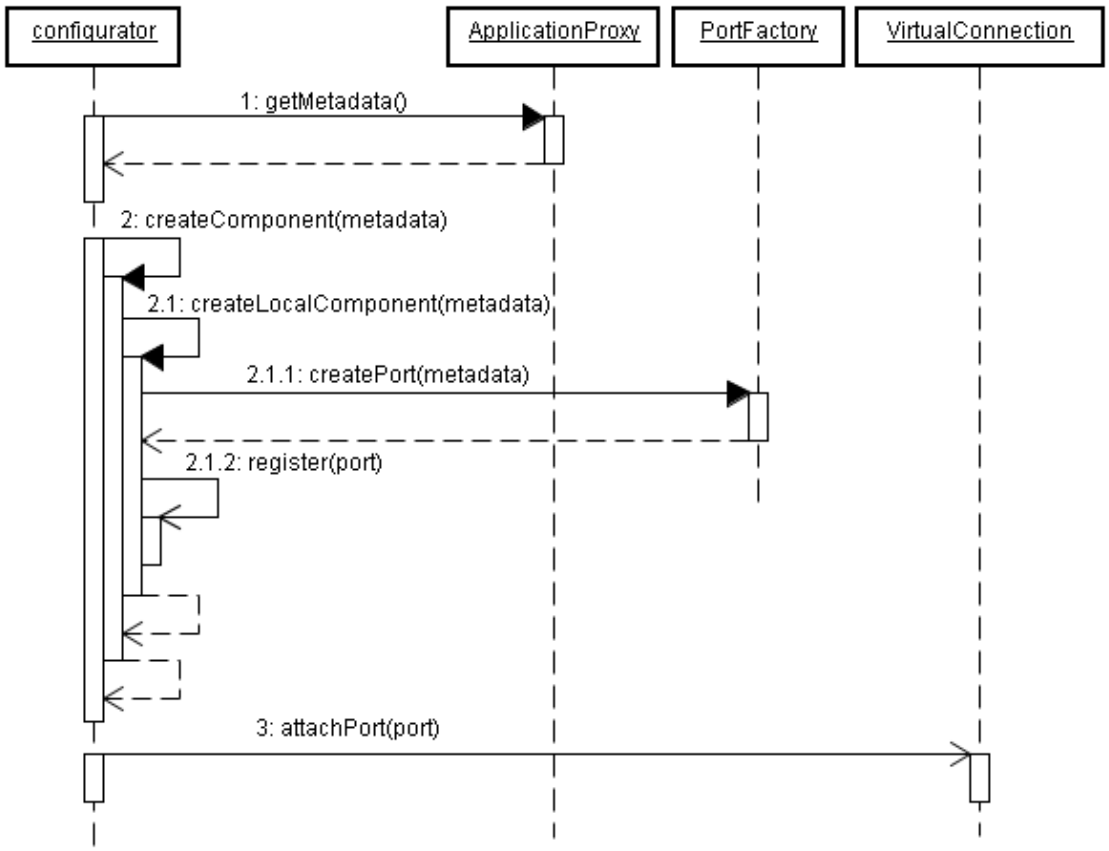

Figura 30. Processo de Criação de uma Porta de Comunicação Local.

Após sua instanciação, a porta é registrada localmente junto ao *Configurator* (*register*), e a referência de sua interface é retornada. Neste momento, a porta de comunicação recém-criada está registrada na lista de recursos do *Configurator*, estando pronta para ser registrada junto à conexão virtual (*VirtualConnection*), através da chamada *attachPort*, para que possa ser configurada juntamente com os elementos que realizam a interconexão.

Este processo é repetido para cada porta envolvida na conexão, com um componente *VirtualConnection* para cada interconexão estabelecida na especificação da aplicação. O processo de criação de uma porta distribuída segue procedimentos semelhantes aos apresentados anteriormente.

Este processo é apresentado na Figura 31. O *MasterConfigurator* recupera os meta-dados correspondentes (*getMetadata*), e realiza a operação *createComponent*, que tem o papel de identificar a localização da porta a ser criada. No caso distribuído, o *MasterConfigurator* realiza a comunicação com o *SlaveConfigurator* solicitando a criação de um componente (*createRemoteComponent*). O *SlaveConfigurator* realiza a chamada *createLocalComponent*, que identifica a fábrica correspondente ao componente a ser criado. Após sua criação pelo elemento *PortFactory* (*createPort*), o *SlaveConfigurator* realiza o registro da porta recém-criada localmente.

Após a aquisição da referência da interface da porta de comunicação, por parte do *MasterConfigurator*, a mesma é passada para o *VirtualConnection*, representada pela chamada *attachPort*.

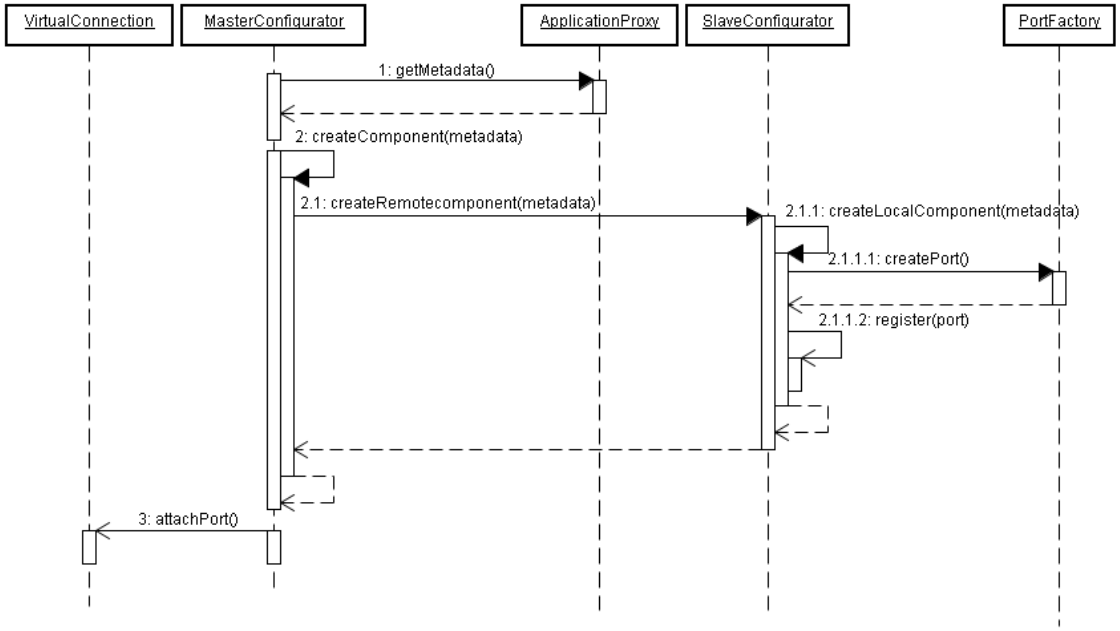

Figura 31. Processo de Criação de uma Porta de Comunicação Distribuída.

De maneira semelhante à apresentada na Figura 29, a criação de um componente remoto exige a interação entre os *Configurators* das plataformas envolvidas. Esta interação é realizada através da tecnologia de distribuição definida pelo *middleware* que implementa o *framework*. Após a criação da porta de comunicação, por parte do

SlaveConfigurator, sua referência remota é retornada para o *MasterConfigurator*, que realiza o registro desta porta junto à conexão virtual correspondente.

3.4.3. Criação dos Canais de Comunicação

Como citado anteriormente, o componente *CommunicationChannel* é formado através da composição de dois subcanais, *SourceSubChannel* e *TargetSubChannel*, e um gerente, *ChannelManager*. O componente *ChannelManager* funciona como ponto de acesso utilizado pela conexão virtual para gerenciar os elementos de um canal de comunicação através da interface *IChannel*, enquanto que os subcanais representam as extremidades de um canal de comunicação.

A Figura 32 apresenta o processo de instanciação de um canal de comunicação para o caso local, onde todos os componentes envolvidos residem no mesmo espaço de endereçamento.

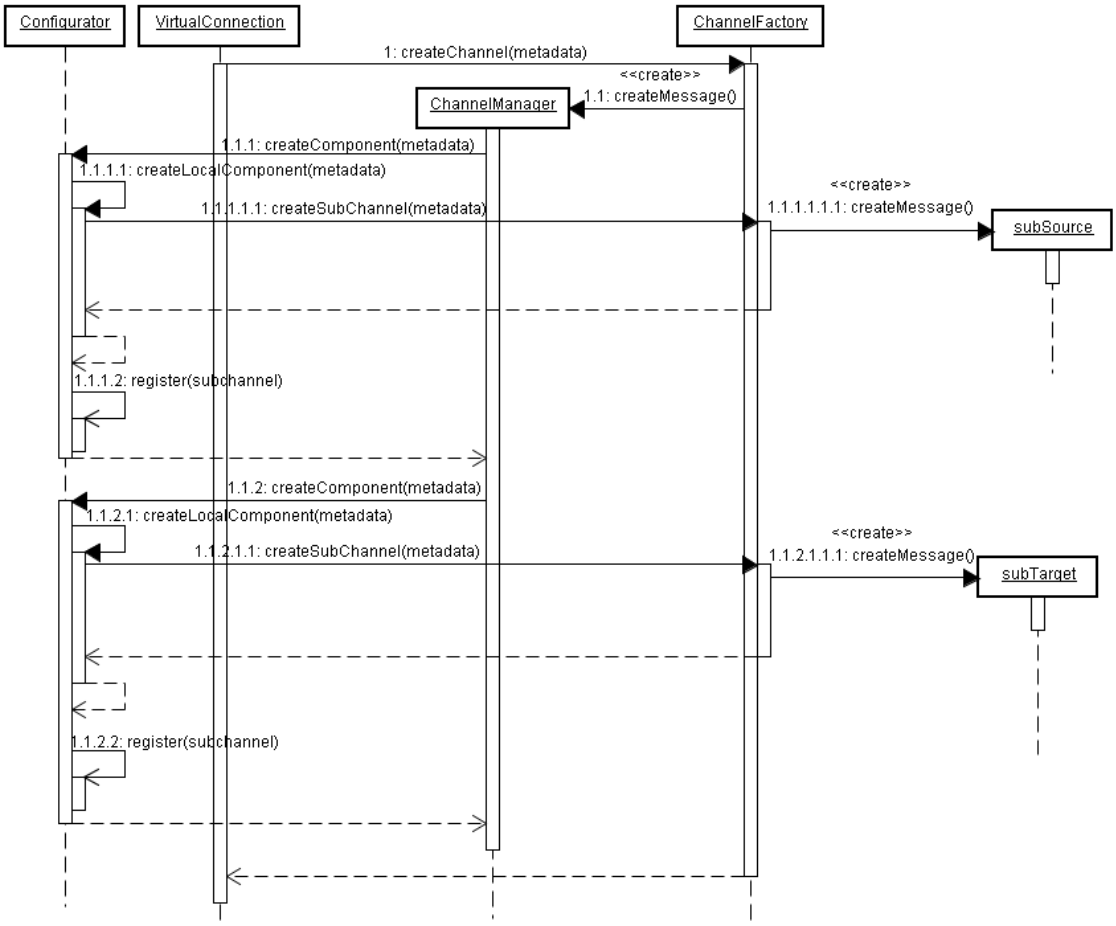

Figura 32. Processo de Criação de um Canal de Comunicação Local.

O canal de comunicação é criado durante o estabelecimento da interconexão pela conexão virtual. Uma vez que o canal é formado por outros componentes, e o componente *ChannelManager* funciona como gerente do canal de comunicação, este componente não é registrado junto ao configurador.

A conexão virtual solicita à fábrica responsável a criação de um canal de comunicação, representado pela chamada *createChannel*, passando como parâmetro os meta-dados referentes aos elementos envolvidos. A fábrica decide a tecnologia de comunicação a ser utilizada baseada nos meta-dados recebidos como parâmetro e nos critérios de escolha implementados, e inicia o processo de criação do canal correspondente, instanciando o componente *ChannelManager*. O componente *ChannelManager* solicita ao seu configurador local a criação de suas extremidades através da operação *createComponent*. Toda a comunicação se dará entre estas extremidades, utilizando a tecnologia que o canal representa para este fim.

O processo de instanciação de um subcanal é semelhante ao processo de instanciação de um componente local apresentado na Figura 28, com uma variação na fábrica ativada para a efetiva instanciação do componente requerido. Primeiramente, a localização do componente a ser criado é determinada baseada nos meta-dados recebidos como parâmetro. No caso local, a operação *createLocalComponent* identifica a fábrica adequada (*ChannelFactory*), que é ativada para a criação do componente identificado (*createSubChannel*).

Os subcanais, que são os elementos ativos em um canal de comunicação e representam as extremidades do mesmo, precisam ser registrados junto ao configurador de forma que os recursos necessários para o desempenho de suas funções sejam gerenciados pelo configurador. Após o registro junto ao *Configurator* (*register*), as referências das interfaces dos subcanais são então retornadas para o *ChannelManager*. Este processo ocorre para cada um dos subcanais.

A Figura 33 apresenta o processo de instanciação de um canal de comunicação para um caso distribuído, com uma extremidade no domínio do *MasterConfigurator* e a outra no domínio do *SlaveConfigurator*. A instanciação do subcanal local acontece de maneira semelhante à apresentada anteriormente.

A operação *createComponent* identifica a localização do componente em questão, enquanto que a operação *createLocalComponent* identifica a fábrica correspondente ao componente indicado pelos meta-dados. Após a instanciação do

componente pela fabrica, o mesmo é registrado junto ao configurador responsável pelo seu espaço de endereçamento (*MasterConfigurator*).

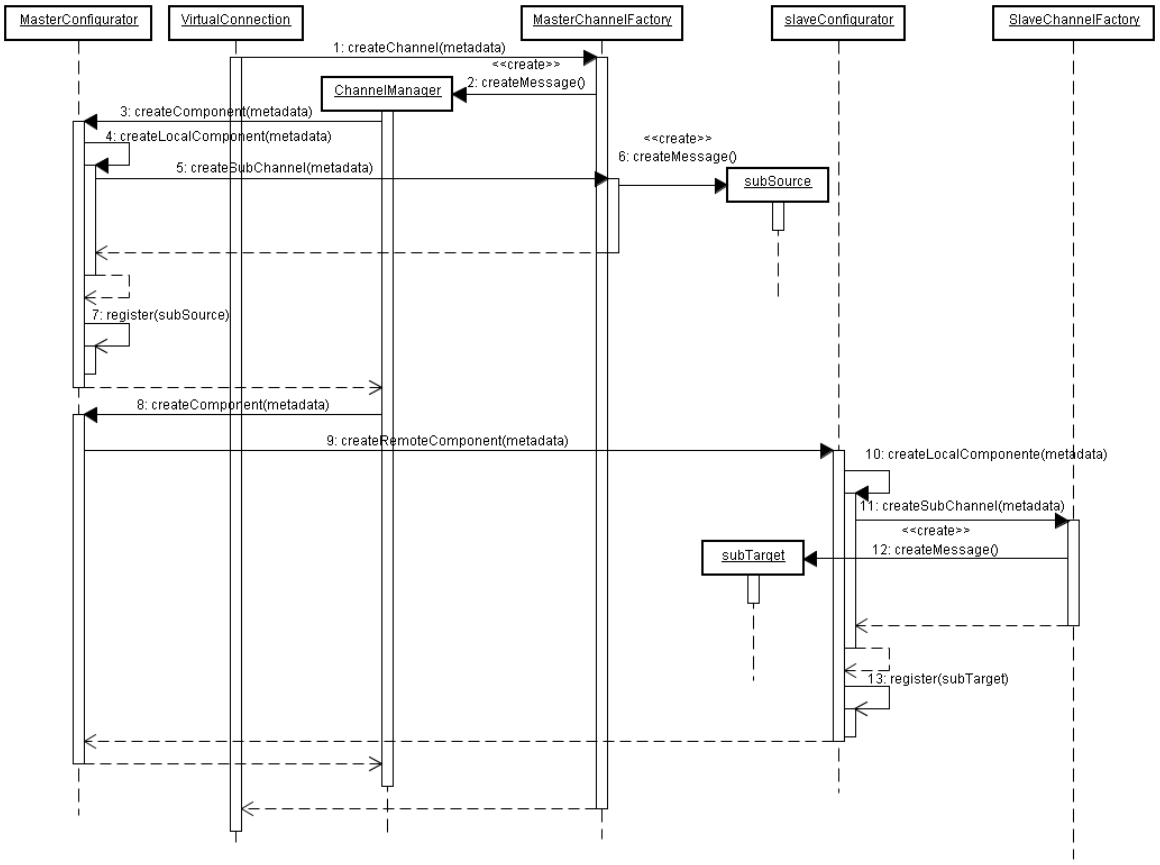

Figura 33. Processo de Instanciação de um Canal Distribuído.

A instanciação do subcanal que se encontra em outro espaço de endereçamento necessita da interação entre os *Configurators*. Após a identificação da localização do subcanal (*createComponent*), o *MasterConfigurator* interage com o *SlaveConfigurator* solicitando a criação de um componente remoto (*createRemoteComponent*). O funcionamento do *SlaveConfigurator* é o mesmo apresentado para a criação de um componente local, diferenciando na fábrica ativada para a criação do componente. O *SlaveConfigurator* utiliza a operação *createLocalComponent* para identificar a fábrica adequada, e após a instanciação do subcanal, realiza o registro local (*register*) do componente recém-criado. Em seguida a referência para uma interface deste componente é retornada para o *MasterConfigurator*.

3.4.4. Configuração Canal-Porta

O processo de instanciação de um canal de comunicação foi apresentado na Figura 32. Este processo é seguido do processo de configuração das portas de comunicação associadas com o canal em questão.

O processo de configuração de um canal de comunicação e suas respectivas portas é exibido na Figura 34, onde a seqüência de operações relacionada com a instanciação dos componentes que compõem um canal de comunicação foi abstraída de forma a simplificar a figura. O processo de instanciação de um canal de comunicação e de seus respectivos subcanais foi abstraído através da chamada *createMessage*.

Após a instanciação do canal, o mesmo é configurado para trabalhar em conjunto com as portas. Para isso, os canais são associados às suas respectivas portas através de chamadas *attachChannel*. Este processo indica à porta os meta-dados referentes ao canal de comunicação que ela vai utilizar, cabendo à porta solicitar a referência do canal de comunicação ao seu configurador local, utilizando-se dos meta-dados recebidos anteriormente (*getChannel*).

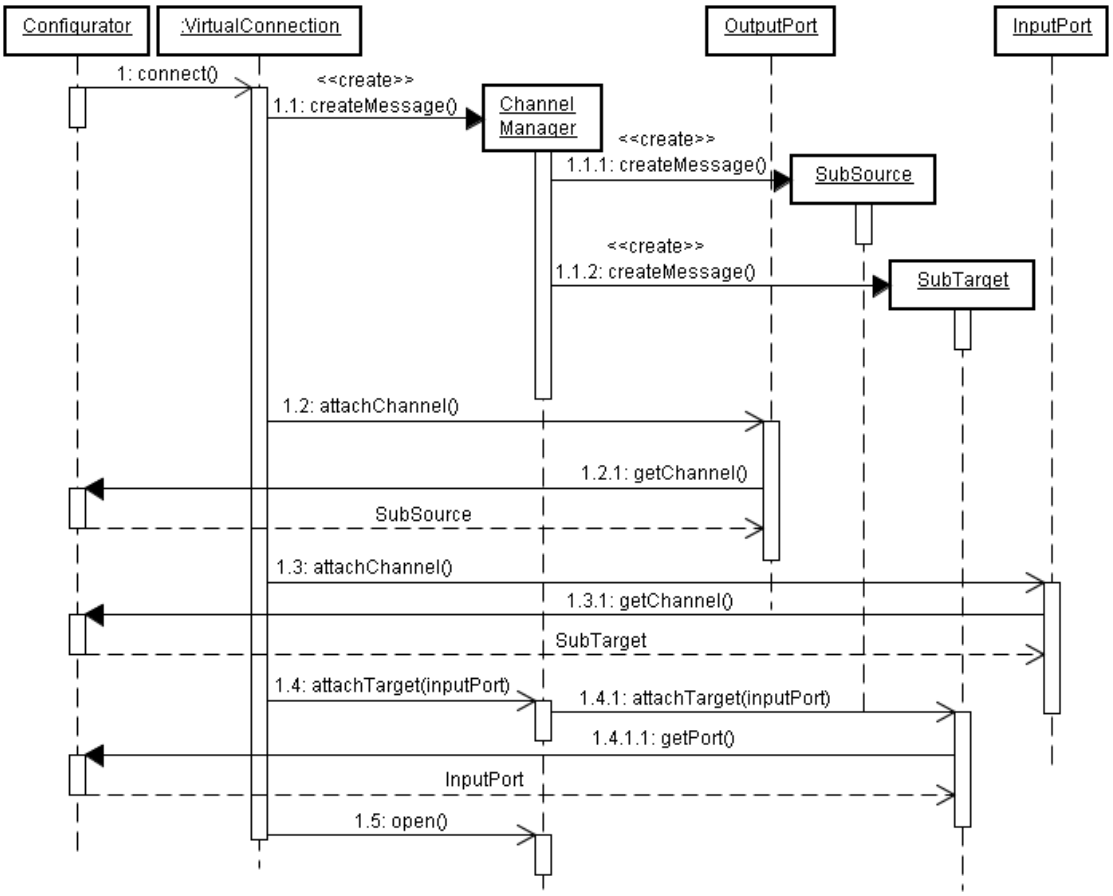

Figura 34. Processo de Configuração de um Canal de Comunicação e suas Portas.

O configurador retorna uma referência a uma interface do canal de comunicação criado anteriormente e alocado para a porta de comunicação. Com isso, consegue-se uma abstração acerca do tipo de canal de comunicação que está sendo utilizado no momento e, por conseguinte, uma abstração da localização das portas e de seus respectivos componentes.

Após a configuração das portas relacionadas ao canal de comunicação em questão, o mesmo deve adquirir a referência para a interface da porta de entrada correspondente. Isto acontece através da chamada *attachTarget*, onde o canal recebe como parâmetro o identificador referente à porta de entrada associada. Esta chamada é encaminhada para a extremidade que trabalha em conjunto com a porta de entrada utilizando a referência adquirida durante o processo de instanciação do canal de comunicação, que por sua vez, solicita ao seu configurador local uma referência da porta de entrada identificada pelo seu parâmetro (*getPort*).

Ao término da configuração dos canais e portas, o componente *VirtualConnection*, através da chamada *open* no canal de comunicação, inicia o processo de ligação entre os componentes que compõem um canal de comunicação. A forma como os subcanais se comunicam depende da tecnologia de comunicação que o canal representa; por exemplo, no caso de uma comunicação local, os subcanais podem se comunicar através de uma referência direta.

O diagrama da Figura 34 apresenta de forma simplificada o processo de configuração dos elementos de uma interconexão. Este processo abstrai a localização dos componentes, uma vez que os componentes utilizam seu configurador local para adquirir as referências das interfaces dos outros elementos, contemplando tanto o caso local quanto o caso distribuído.

No caso distribuído, os componentes que se encontram fora do espaço de endereçamento do *Configurator* que processou a especificação se reportam ao seu *Configurator* local, de maneira a adquirir as referências indicadas.

3.4.5. Obtenção das Portas de Comunicação por Parte dos Recursos Virtuais

Da mesma maneira que as portas de comunicação se reportam ao seu configurador local para recuperar as referências dos canais de comunicação envolvidos, e os subcanais para recuperarem as referências das portas, os recursos também o fazem para recuperar suas respectivas portas de comunicação.

O diagrama da Figura 35 apresenta um exemplo do processo de aquisição, por parte dos recursos virtuais, de suas portas de comunicação. Este exemplo envolve a participação de um componente produtor e um componente consumidor, representados pelos elementos *Producer* e *Consumer*, respectivamente.

Os recursos envolvidos neste exemplo estão em um ambiente distribuído com o elemento *ConfiguratorSource* representando o configurador responsável pela plataforma onde se encontra o elemento *Producer* e o *ConfiguratorTarget* representando o configurador da plataforma onde se encontra o elemento *Consumer*. A chamada *connect* representa o processo de interconexão realizada pela conexão virtual, onde os elementos da interconexão são instanciados e configurados. As referências para os recursos são adquiridas no momento de sua instanciação, quando o *Configurator* realiza o registro junto ao *ApplicationProxy* destas referências.

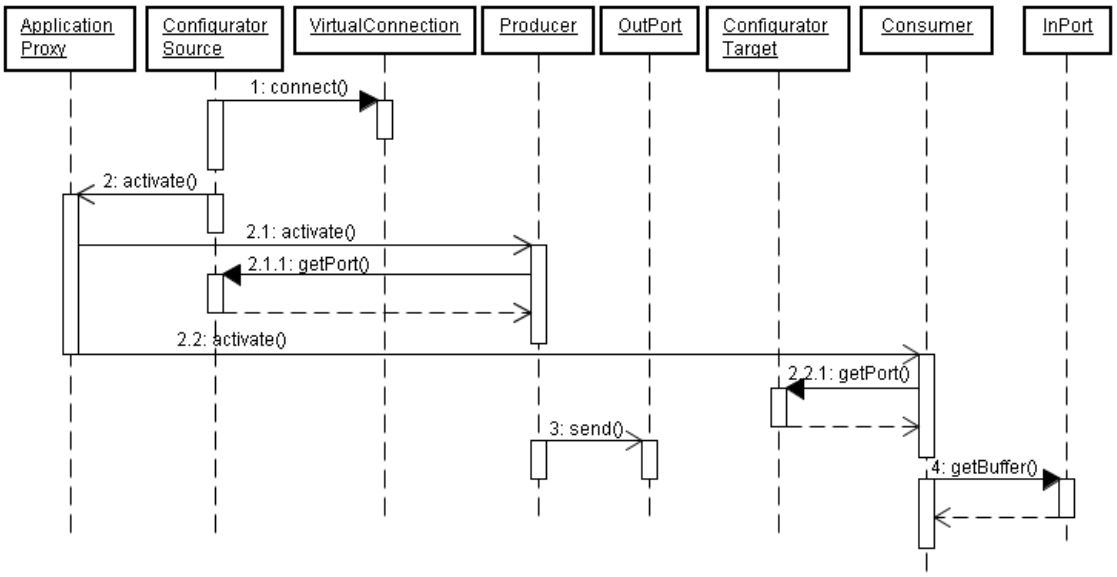

Figura 35. Processo de aquisição das portas de comunicação por parte dos recursos virtuais.

Para iniciar a aplicação, o Configurador mestre, representado pelo elemento *ConfiguratorSource*, realiza uma chamada de ativação junto ao *ApplicationProxy* correspondente (*activate*). O *ApplicationProxy* notifica todos os recursos da aplicação para iniciarem suas atividades (*activate*), utilizando as referências que foram registradas pelo *Configurator* no momento da instanciação dos recursos. Os recursos utilizam seu *Configurator* local, *ConfiguratorSource* para o componente *Producer*, e *ConfiguratorTarget* para o componente *Consumer*, para a aquisição da referência da porta de comunicação alocada para eles.

Esta aquisição esta representada pela chamada *getPort* por parte dos recursos para seus respectivos *Configurators*. Após a aquisição das portas de comunicação os recursos iniciam seu funcionamento, transmitindo ou recebendo dados.

3.4.6. Adaptação

O modelo de interconexão proposto oferece suporte a adaptação reativa e pró-ativa conforme definido pelo *framework* Cosmos. Em uma adaptação reativa, a aplicação inicia o processo de adaptação, solicitando a alteração de propriedades de componentes da aplicação. Esta solicitação pode partir, por exemplo, de uma interação com o usuário. Na adaptação pró-ativa, os requisitos de QoS e as políticas de adaptação são descritas juntamente com a especificação da aplicação, conforme definido pelo modelo de gerenciamento de QoS apresentado em [31]. Neste caso, monitores iniciam o processo de adaptação ao detectarem parâmetros de QoS definidos na especificação fora da faixa de valores aceitáveis especificados.

Do ponto de vista do modelo de interconexão, o elemento que iniciou a adaptação não interfere na seqüência de operações desempenhadas durante o processo. Nos dois tipos de adaptação citados, o *Configurator* é acionado para realizar o processo de adaptação, verificando a consistência da nova configuração antes de efetivamente alterar a configuração dos componentes da aplicação.

A Figura 36 apresenta, de maneira simplificada, os principais passos de um processo de adaptação.

Ao ser acionado para iniciar um processo de adaptação, o configurador precisa analisar a solicitação de forma a não deixar o sistema em um estado inconsistente, representado pela chamada *validate*. Para isso, o configurador identifica os componentes afetados por esta mudança, iniciando um novo processo de negociação para verificar se estes componentes suportam a alteração solicitada.

Após essa verificação, o configurador efetivamente inicia o processo de adaptação notificando os componentes envolvidos para que estes iniciem uma etapa de reserva de recursos e se preparem para a reconfiguração, representada pela chamada *reserveResources*. Esta reserva de recursos acontece de forma paralela com a execução do componente.

O *Configurator* notifica então o correspondente componente *VirtualConnection*, informando-o que um processo de adaptação foi iniciado (*reconfig*). A conexão virtual realiza a mesma operação de alocação de recursos e preparação para a reconfiguração

nas portas associadas (*reserveResource*), para em seguida realizar uma duplicação do canal de comunicação. Com isso, para cada canal existente na conexão, existirá um canal secundário, que será utilizado pelas portas de comunicação para a transmissão do novo fluxo de informações. Isso é feito para garantir a sincronização entre os fluxos, de forma que não aconteça uma interrupção na apresentação para o usuário da aplicação.

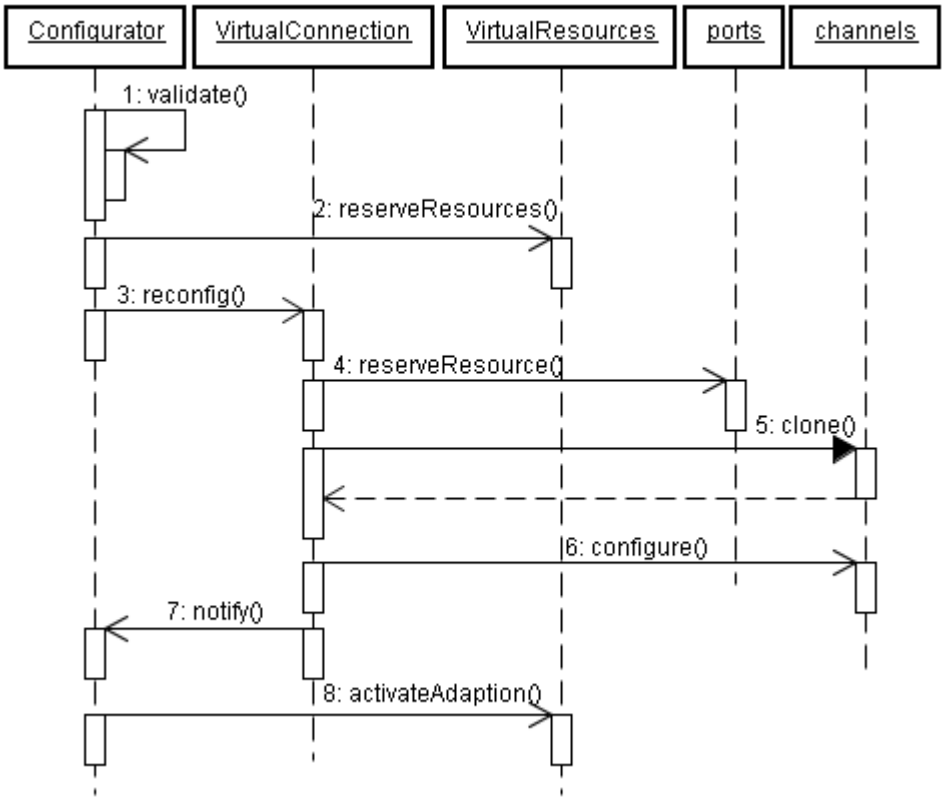

Figura 36. Processo Simplificado de Adaptação.

Quando todos os canais secundários forem instanciados e configurados com as portas de comunicação dos canais originais (*configure*), o fluxo de dados atual é adaptado sem interrupção na transmissão. Ao fim do processo de configuração dos canais secundários, as portas de entrada associadas à conexão estarão prontas para receber dados destes novos canais.

Cada uma das portas envolvidas passa a ter, temporariamente, dois canais de comunicação, continuando a operar através de seu canal primário. Em seguida, a conexão virtual realiza uma notificação para o *Configurator*, informando que o processo de alocação de recursos está concluído. O *Configurator* então notifica os componentes envolvidos que eles podem iniciar a reconfiguração, representado pela chamada *activateAdaption* ao elemento *VirtualResources*.

O processo de adaptação pode ser dividido em duas fases de forma a facilitar seu entendimento: fase de preparação e fase de ativação da adaptação. Durante a fase de preparação, todos os componentes envolvidos são notificados, tendo seu estado de funcionamento alterado de acordo com a máquina de estados definida pelo *framework* Cosmos [4]. Esta alteração de estados contempla o processo de alocação de recursos e preparação para a reconfiguração. A fase de ativação da adaptação contempla as operações que efetivamente realizam a troca dos parâmetros operacionais dos componentes envolvidos, onde os canais de comunicação são trocados, e os recursos virtuais envolvidos alteram suas propriedades.

A Figura 37 apresenta um diagrama de seqüência com as principais operações da fase de preparação. Como explicitado anteriormente, uma adaptação sempre é controlada pelo configurador. O configurador realiza uma negociação de forma a validar a nova configuração (*validate*) e, em caso de validação positiva, inicia o processo de adaptação.

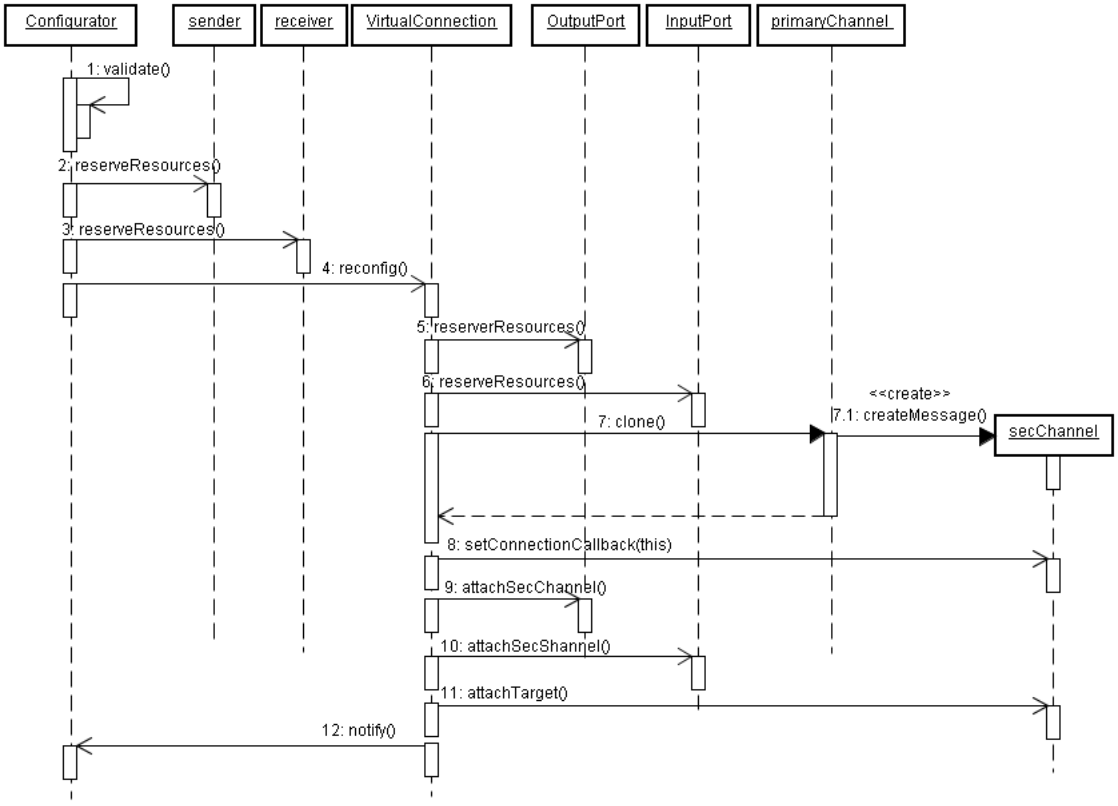

Figura 37. Processo de Adaptação, Fase de Preparação.

O primeiro passo consiste em notificar os componentes envolvidos para que eles possam iniciar o processo de alocação de recursos e se prepararem para a adaptação,

representado pelas chamadas *reserveResources* para os componentes *sender* e *receiver*. Em seguida, o *Configurator* notifica a conexão virtual que inicia o processo de adaptação relacionado com as portas e os canais de comunicação associados (*reconfig*).

As portas são notificadas que uma adaptação está em andamento, iniciando a alocação dos recursos necessários para esta reconfiguração, representadas pelas chamadas *reserveResources* junto às portas de comunicação. Elas se preparam para receber o novo canal, que é o canal secundário da interconexão, alocando um novo *buffer* para este canal, de maneira a não interromper o fluxo original enquanto se prepara para a reconfiguração.

Após a notificação das portas, a conexão virtual inicia o processo de clonagem dos canais de comunicação (*clone*). Vale salientar que o processo de criação de um canal de comunicação secundário é idêntico ao de canais primários, apresentado na Figura 32, e está representado na Figura 37 pela chamada *createMessage* ao elemento *secChannel*.

Com o canal secundário criado, a conexão virtual inicia sua configuração, passando as referências necessárias para o funcionamento do canal (*setConnectionCallback*), e configurando as portas com o novo canal de comunicação, representado pela chamada *attachSecChannel*. De forma análoga à Figura 34, o canal secundário obtém as referências das portas de comunicação associadas.

Após a configuração do canal secundário e das portas de comunicação envolvidas, a conexão virtual notifica o *Configurator* (*notify*), informando-o que o processo de preparação e alocação de recursos para a reconfiguração foi concluído com sucesso e que a reconfiguração pode continuar.

O configurador após receber a notificação da conexão virtual inicia a segunda fase do processo de adaptação apresentado na Figura 38. Primeiramente, os componentes transmissores são identificados e então notificados para que possam ativar a adaptação previamente preparada, representada pela chamada *activateAdaption*. Neste ponto, o componente transmissor interrompe a sua transmissão e altera seus parâmetros operacionais. O componente realiza então uma notificação à sua porta de saída (*reconfig*) e recomeça a transmissão. A porta de saída, por sua vez, realiza a troca do canal de comunicação (*switchChannels*), marcando o canal primário como livre (*free*) e continua a operar normalmente enviando dados, agora, pelo canal secundário. Caso ainda tenha algum dado do fluxo em trânsito no canal de comunicação primário, ele chegará até a porta de entrada, enquanto, paralelamente, a transmissão prossegue,

saindo do componente transmissor através do canal secundário, representado pela seqüência de operações *send* e *put*.

Mesmo considerando que o canal de comunicação primário é liberado pela porta de saída quando ela indica a troca de canais, ele continua existindo até ser liberado pela porta de entrada. Os dois canais, primário e secundário, existem e funcionam em paralelo durante o processo de adaptação. A porta de entrada, que já está com seu buffer de recepção secundário alocado, recebe dados dos dois canais de comunicação existentes, colocando os dados em *buffers* distintos. Neste momento, a porta de entrada possui dois *buffers* de recepção, um para o canal primário e um para o canal secundário.

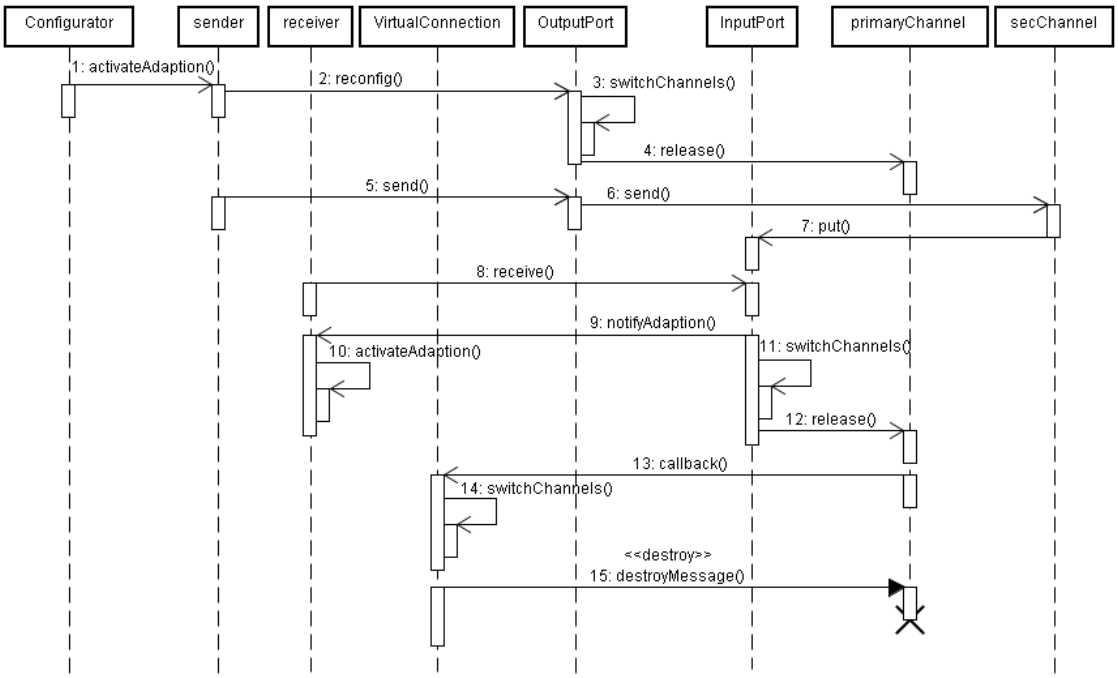

Figura 38. Processo de Adaptação, Fase de Ativação.

O componente receptor continua sua operação solicitando dados da porta de entrada (*receive*). A porta de entrada, por sua vez, continua entregando os dados do *buffer* correspondente ao canal primário, até que o mesmo se esvazie. Quando o buffer primário da porta de entrada não contiver mais dados, assume-se que o fluxo de dados do canal de comunicação primário terminou, e que este canal não irá entregar mais nenhum dado para a porta de entrada.

Vale salientar que o fato do buffer de recepção primário da porta de entrada não ter dados não representa o final do fluxo de dados do canal de comunicação primário. Para isso é necessária a definição de uma estratégia que garanta o fim do fluxo de

dados. Esta estratégia é dependente da tecnologia de comunicação utilizada pelo canal de comunicação. Para o presente trabalho, assumiu-se que o buffer vazio representa o fim do fluxo de dados do canal de comunicação associado.

Neste momento, esta porta notifica seu componente (*notifyAdaption*) e inicia o processo de troca dos canais de comunicação e de seus *buffers* de recepção (*switchChannels*).

Após a notificação, o componente receptor realiza os procedimentos necessários à reconfiguração dinâmica, representado pela chamada *activateAdaption*, trocando seus parâmetros de operação e retornando ao seu funcionamento normal.

No final da troca dos canais, a porta de entrada notifica ao canal primário que ele não é mais necessário, através da chamada *release*. O canal primário, com as notificações de suas duas portas envolvidas realiza uma chamada *callback* à conexão virtual correspondente, informando que o canal primário não está mais sendo utilizado e que o mesmo pode ser desalocado. A conexão virtual realiza a troca dos canais de comunicação (*switchChannels*), fazendo com que o canal de comunicação secundário passe a ser o canal primário da interconexão.

3.4.7. Comunicação entre Configuradores

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. Isto ocorre devido ao fato do configurador ser o elemento responsável pelo gerenciamento dos componentes do seu espaço de endereçamento, cabendo a ele validar ou não uma determinada configuração envolvida em um ambiente distribuído.

Devido a isso, é preciso dar suporte à realização de comunicação entre os configuradores distribuídos envolvidos. Esta comunicação trata questões relacionadas com a alocação de recursos, negociação de propriedades e instanciação de componentes remotos.

De forma a permitir esta interação entre configuradores remotos, foi definida a interface *IRemoteConfiguration*. Esta interface é apresentada na Figura 39, e oferece a operação *createRemoteComponent* para solicitar a criação de um componente, recebendo como parâmetro os meta-dados do componente solicitado.

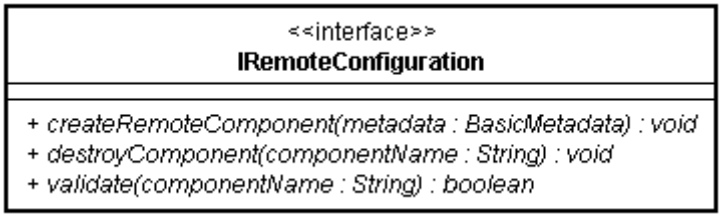

Figura 39. Interface IRemoteConfiguration.

De maneira análoga, a operação *destroyComponent* é utilizada para solicitar a remoção de um componente. A operação *validate* é utilizada para validar uma determinada configuração junto ao *Configurator* remoto. Uma vez que a configuração é validada, o *Configurator* escravo já deixa os recursos necessários alocados. Caso este pedido de validação tenha uma resposta negativa, o *Configurator* mestre deve alterar a configuração da aplicação e realizar uma nova tentativa de validação junto ao *Configurator* escravo.

3.5. Considerações Finais

O modelo de interconexão proposto tem o objetivo de tratar os aspectos de comunicação entre componentes multimídia no contexto do *framework* Cosmos. Ele explora os conceitos de porta de comunicação e conexão virtual utilizados pelo *framework* de forma a prover uma arquitetura genérica e flexível para a troca de fluxo de dados multimídia entre componentes em ambientes multimídia distribuídos e heterogêneos.

O conceito de portas de comunicação tem sido largamente utilizado pela engenharia de software, consituindo-se num elemento de grande flexibilidade ao permitir tratar conexões *unicast* e *multicast* com abstrações similares, permitindo ao modelo suportar diversas topologias de interconexão. Uma conexão virtual abstrai os detalhes dos mecanismos de comunicação utilizados, oferecendo uma interface bem definida para o gerenciamento de uma interconexão.

O modelo explora os conceitos de propriedades e reflexividade presentes no *framework* Cosmos ao oferecer o suporte à realização de reconfigurações dinâmicas nos componentes da aplicação.

4. Uma Implementação Piloto

Com o objetivo de avaliar os conceitos e a arquitetura do *framework* Cosmos, foi iniciado um processo de instanciação do mesmo para sistemas de TVDI, culminando com a definição de uma arquitetura conceitual de um *middleware* adaptativo e reflexivo denominado AdapTV [25].

O AdapTV é um *middleware* baseado no Cosmos que provê uma infra-estrutura adaptativa de suporte a execução, gerenciamento e configuração de componentes e aplicações de sistemas de televisão digital interativa [32]. O protótipo inicial deste *middleware* serviu como prova de conceito para o *framework* Cosmos, tendo sua implementação realizada utilizando-se a linguagem de programação Java [33], onde os componentes foram implementados por classes Java. Seu desenvolvimento foi concentrado nos conceitos relacionados com a negociação de propriedades e adaptação com uma abordagem estática.

Como prova de conceito para o modelo de interconexão apresentado neste trabalho, foi realizado uma extensão do protótipo inicial do *middleware* AdapTV, de forma a incorporar os conceitos introduzidos pelo modelo de interconexão. Neste novo protótipo foram realizadas instanciações e testes envolvendo a interconexão de componentes recursos virtuais, primeiramente em um ambiente de execução local, e, em seguida, em um ambiente distribuído. Além disso, foram realizados também testes de adaptação dinâmica reativa e pró-ativa, envolvendo os componentes do modelo de QoS definido [31].

O presente capítulo explora a arquitetura de implementação do modelo de interconexão de componentes e sua incorporação ao protótipo inicial do *middleware* AdapTV [32]. No capítulo são descritas as classes definidas no escopo do presente trabalho, e apresentadas algumas aplicações exemplo definidas para validar os conceitos do Cosmos e do modelo de interconexão proposto.

4.1. Mapeamento do Modelo

Esta seção apresenta o mapeamento dos componentes do modelo para as classes definidas na implementação do protótipo. As principais classes definidas são apresentadas na Figura 40, juntamente com o relacionamento entre elas.

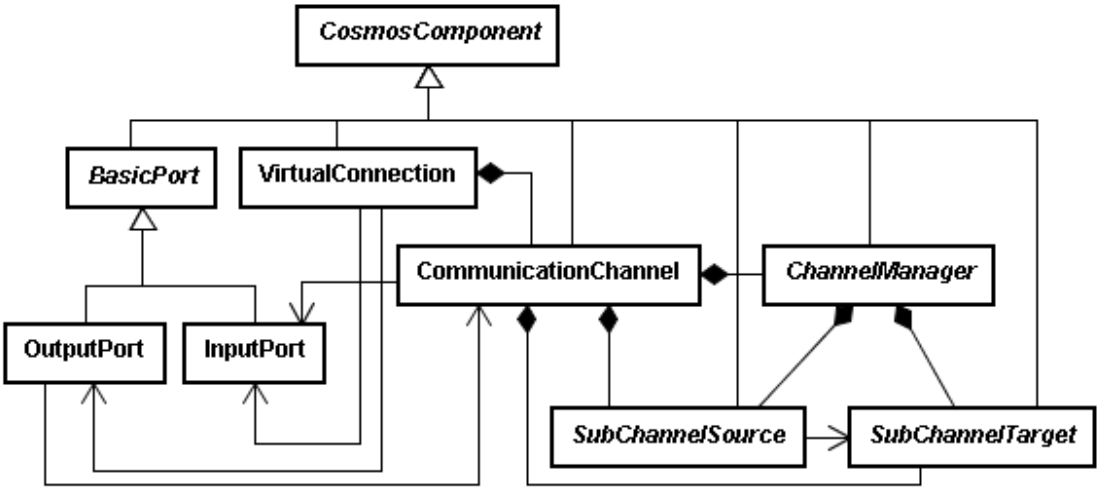

Figura 40. Principais classes do modelo

O *framework* Cosmos define um conjunto de interfaces que dão suporte aos conceitos de propriedades e reflexividade [4]. Estas interfaces são chamadas de interfaces básicas, e devem ser oferecidas por todos os componentes do modelo. Para isso, foi definida uma classe abstrata chamada *CosmosComponent* que agrega todas estas interfaces em um só elemento, facilitando a construção de componentes customizados, bastando para isso estender a classe *CosmosComponent*. A definição destas interfaces está fora do escopo deste trabalho sendo descritas em [4]. Para o presente trabalho, a abordagem considera que os elementos definidos no modelo de interconexão são tratados como componentes do *framework*, devendo, portanto, oferecer as interfaces básicas, suportando a utilização de propriedades e configuração. A Figura 40 ilustra a classe *CosmosComponent* como a classe pai de todas as outras classes do modelo.

Para a implementação da porta de comunicação, foi definida uma classe abstrata para tratar as funcionalidades comuns às portas, independentemente do seu tipo efetivo ser caracterizado como entrada ou saída. A definição de um tipo de porta específico é obtida através da herança desta classe abstrata. Uma conexão virtual é representada pela

classe *VirtualConnection*, que mantém referências para portas de entrada e portas de saída.

O canal de comunicação é formado através de uma composição com outros três componentes. Estes componentes são os subcanais que representam as extremidades de um canal de comunicação e o *ChannelManager*, que gerencia o canal. Na implementação, cada componente de um canal de comunicação foi mapeado para uma classe abstrata que caracteriza as operações de um *ChannelManager* ou de um dos dois subcanais.

A classe *CommunicationChannel* representa o componente canal de comunicação, sendo formada pelas outras classes que representam os respectivos subcomponentes de um canal de comunicação. A classe *ChannelManager* é a classe responsável por manter as referências dos subcanais. As chamadas realizadas pela conexão virtual são encaminhadas para serem atendidas pela classe *ChannelManager*. Os subcanais realizam a comunicação com as portas e entre as mesmas. Uma porta de saída envia dados que são encaminhados para os subcanais. Chamadas oriundas das portas de comunicação em direção ao canal de comunicação são encaminhadas até a classe *ChannelManager*.

A Figura 41 apresenta as classes definidas para a implementação das portas de comunicação. Uma conexão virtual oferece a interface *IVirtualConnection*, que é utilizada pelo *Configurator* para gerenciar uma interconexão. Esta interface é implementada pela classe *VirtualConnection* para gerenciar as portas e os canais de comunicação envolvidos.

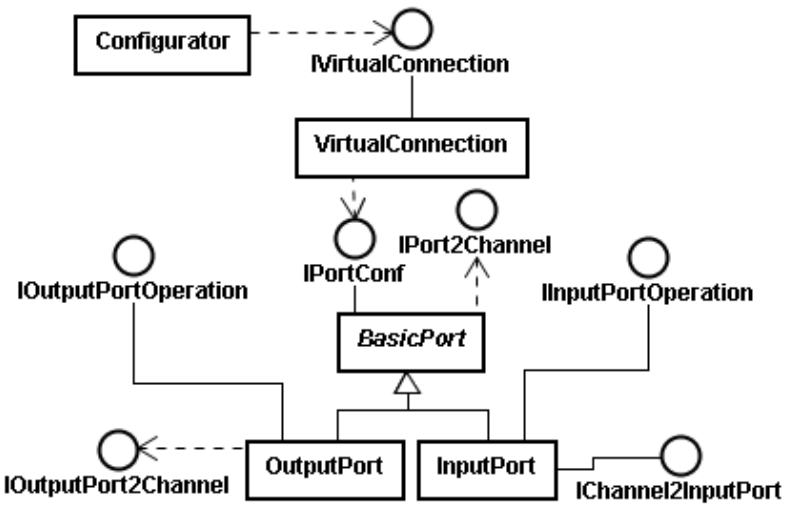

Figura 41. Classes para implementação das portas de comunicação.

As portas de comunicação são configuradas através da interface *IPortConf* pela conexão virtual, sendo utilizada por exemplo, para iniciar uma adaptação. Para isso, foi definido a classe abstrata *BasicPort* que implementa esta interface e funciona como uma abstração para representar as portas. A interface *IPort2Channel* é acessada pelas portas para sinalizar a um canal que o mesmo já pode ser descartado.

As classes *OutputPort* e *InputPort* representam as portas de saída e de entrada, respectivamente. Uma porta de saída utiliza a interface *IOutputPort2Channel* para acessar o canal de comunicação e oferece a interface *IOutputPortOperation*, que é utilizada pelos recursos virtuais para o envio de dados. A porta de entrada oferece a interface *IInputPortOperation*, utilizada pelos recursos para o recebimento de dados e, a interface *IChannel2InputPort*, utilizada pelo canal de comunicação para a entrega dos dados provenientes da interconexão.

A Figura 42 apresenta um diagrama de classes no contexto do canal de comunicação, onde são explicitadas as classes definidas na sua implementação. A conexão virtual utiliza a interface *IChannel* para gerenciar os canais de comunicação. A classe abstrata *ChannelManager* tem o objetivo de garantir a incorporação, por parte de um canal de comunicação, dos métodos de gerenciamento definidos na interface *IChannel*. Esta classe permite representar diferentes tecnologias de comunicação para os canais de comunicação de maneira transparente.

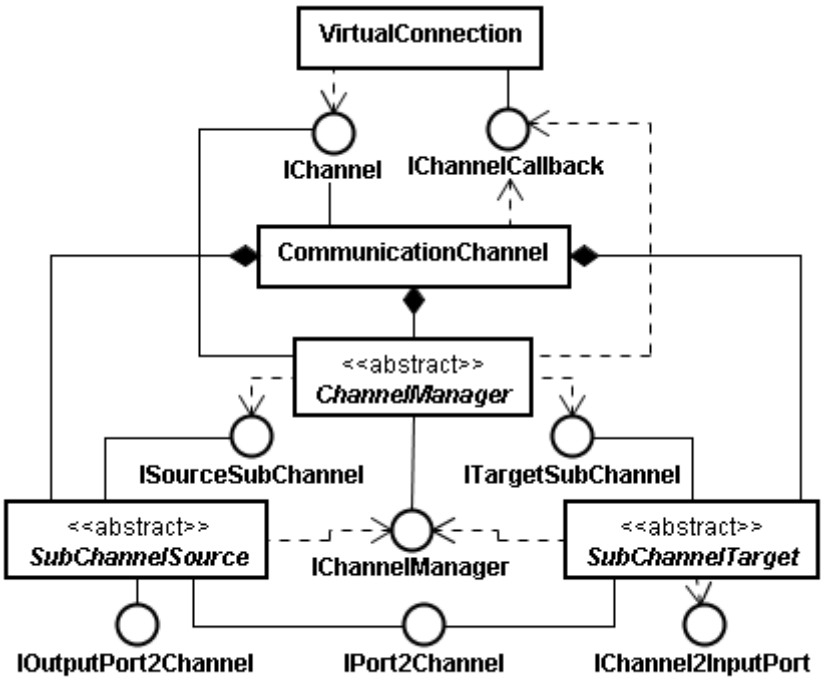

Figura 42. Classes abstratas que compõem um canal de comunicação.

Um canal de comunicação possui dois subcanais que representam suas extremidades. Na implementação, foi definida uma classe abstrata para cada um destes elementos. A comunicação entre o canal de comunicação e seus subcanais se da através das interfaces *ISourceSubChannel e ITargetSubChannel*, no sentido canal-subcanal e, *IChannelManager*, no sentido contrário. O canal de comunicação configura seus subcanais de acordo com as configurações e operações ativadas pela conexão virtual através da interface *IChannel*.

A interface *IOutputPort2Channel*, utilizada pela porta de saída para o envio de dados, é oferecida pelo subcanal que recebe dados do lado transmissor na interconexão, sendo oferecida pela classe abstrata *SubChannelSource*. A classe abstrata *SubChannelTarget* representa a extremidade receptora de um canal de comunicação; ela utiliza a interface *IChannel2InputPort* para a entrega de dados junto à porta de entrada. A interface *IPort2Channel* é oferecida pelas classes abstratas que representam os subcanais, e é utilizada pelas portas de comunicação para sinalizar a um canal que o mesmo não é mais necessário. Conforme mencionado anteriormente, isto acontece durante uma adaptação. Neste caso, os subcanais notificam o *ChannelManager* correspondente, que, após a confirmação de seus dois subcanais, dispara uma notificação à conexão virtual através da interface *IChannelCallback*.

Explorando o canal de comunicação, o modelo permite a utilização de diferentes tecnologias de comunicação. Com a utilização de classes abstratas para representação dos componentes que fazem parte de um canal de comunicação, garante-se a flexibilidade oferecida pelo canal de comunicação, bastando ao desenvolvedor estender as classes abstratas que formam um canal de comunicação para implementar uma tecnologia de comunicação em particular.

Em seguida, é apresentado um exemplo de implementação com uma hierarquia de classes concretas para o caso de um canal de comunicação local, que realiza a comunicação entre seus subcanais através de referências diretas. A Figura 43 apresenta um diagrama de classes ilustrando esta hierarquia para o caso de um canal local. Através da herança da classe abstrata *ChannelManager*, a classe *LocalChannel* tem acesso e implementa as interfaces necessárias para a comunicação com seus subcanais, *LocalSource* e *LocalTarget*. Os subcanais herdam das classes abstratas que representam os subcanais, tendo acesso às interfaces que permitem a comunicação com o canal e às interfaces de comunicação com as portas de comunicação, seja provendo a interface que será usada pela porta de saída, *IOutputPort2Channel*, oferecida pela classe abstrata

SubChannelSource, ou acesso à interface *IChannel2InputPort* oferecida pela porta de entrada para a recepção de dados.

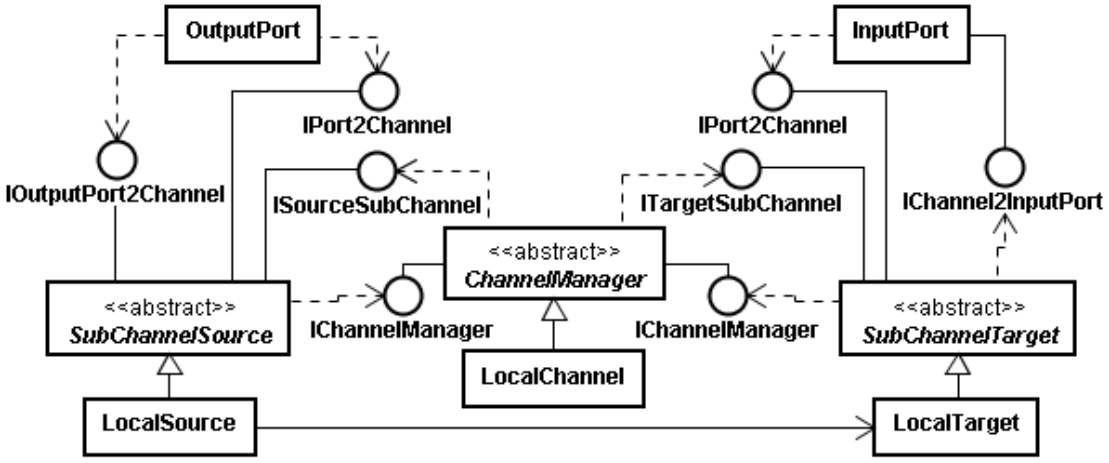

Figura 43. Hierarquia de classes de um canal de comunicação local.

Os subcanais implementam a troca efetiva de dados utilizando uma dentre as diversas tecnologias existentes para este fim. No caso de um canal local, a implementação utiliza uma referência direta para a troca de dados entre os subcanais. Em uma conexão remota, utilizando, por exemplo, o protocolo UDP, os subcanais se comunicam através de um *socket* UDP. O controle dos subcanais é realizado pela classe que herda da classe abstrata *ChannelManager*, *LocalChannel* no caso da implementação do canal local.

Devido à possibilidade da utilização de diferentes tecnologias de comunicação, inclusive de forma simultânea, o desenvolvedor deve estabelecer critérios para a escolha da tecnologia a ser empregada no canal de comunicação. Na implementação do protótipo foram implementados dois tipos de canais de comunicação: canal local e canal UDP. O critério para a escolha da tecnologia de comunicação empregada é baseado na localização dos componentes envolvidos. Os componentes podem estar sendo executados em uma mesma máquina virtual, em uma mesma máquina física, mas em máquinas virtuais diferentes ou em duas máquinas físicas distintas.

4.1.1. Distribuição dos Componentes

A escolha da tecnologia utilizada para a transparência de localização dos componentes do *framework* fica a critério do *middleware* que implementa o *framework*, devendo realizar as alterações necessárias no componente *Configurator* para incorporar

a tecnologia utilizada. Com isso, somente o *Configurator* toma conhecimento da tecnologia de distribuição utilizada, garantindo a transparência para os outros componentes do *framework*.

Devido ao fato da implementação do protótipo ter sido realizada utilizando a linguagem de programação Java, decidiu-se pelo uso da invocação de métodos remotos para a distribuição dos componentes através da tecnologia Java RMI [18]. As mudanças realizadas no protótipo dizem respeito à incorporação da tecnologia Java RMI ao código existente, fazendo com que as classes que representam os componentes remotos incorporassem a interface *Remote*, e as interfaces que serão utilizadas para o acesso remoto precisam ter seus métodos alterados, para que disparem a exceção *RemoteException*, conforme definido pelo Java RMI.

O configurador é o componente responsável pelo controle dos recursos da plataforma. Ele realiza a negociação de propriedades e a instanciação dos componentes que fazem parte da aplicação e do *middleware*. Devido a isso, deve existir uma comunicação entre os configuradores responsáveis pelas plataformas envolvidas. Para que ocorra esta comunicação, a interface *IRemoteConfiguration* foi alterada para que atendesse aos requisitos da tecnologia Java RMI.

O *ApplicationProxy* funciona como um repositório de meta-informações, devendo ser acessível remotamente, de forma que configuradores, ou componentes, remotos consigam obter informações sobre a aplicação e sobre os componentes que estão em seu espaço de endereçamento. Conforme apresentado em [4], o acesso ao *ApplicationProxy* se dá através da interface *IIntrospection*, tendo sido a mesma alterada para atender aos requisitos impostos pela tecnologia de distribuição utilizada.

O componente recurso virtual foi mapeado para a classe abstrata *VirtualResource* que incorpora as interfaces introduzidas pelo modelo de recurso virtual apresentado em [34]. Os recursos ficam registrados no *ApplicationProxy* que reside no espaço de endereçamento do configurador que realizou o processamento da especificação da aplicação. A alteração de uma propriedade de um componente é tratada pelo configurador através do *ApplicationProxy*, de onde o configurador recupera os meta-dados que descrevem a aplicação, de forma a validar novas configurações. A alteração das propriedades de um recurso se dá através da interface *IPropertyUpdate* utilizada pelo *ApplicationProxy* para o envio de comandos aos componentes da aplicação. Para o suporte ao funcionamento distribuído, a implementação desta interface incorpora a tecnologia Java RMI.

Para definir um novo recurso basta utilizar herança a partir da classe abstrata *VirtualResource*, incorporando assim as interfaces oferecidas por um recurso virtual. Algumas operações podem ter funcionamentos variados a depender do recurso em questão, definindo assim um conjunto de métodos abstratos a serem implementados pelo desenvolvedor do recurso. As operações comuns a um recurso virtual são implementadas pela classe abstrata *VirtualResource*.

Assim como um recurso, a porta de comunicação também pode ser gerenciada de maneira remota. Este gerenciamento é realizado pela conexão virtual, através da interface *IPortConf*, que deve incorporar as mudanças introduzidas pela distribuição do protótipo.

Um canal de comunicação é formado por dois subcanais, cada um residindo no espaço de endereçamento da porta de comunicação associada. A comunicação entre o canal e seus subcanais se dá através das interfaces apresentada anteriormente na Figura 25, bastando que estas interfaces incorporem as mudanças necessárias ao funcionamento distribuído.

Além das mudanças nas interfaces dos componentes, foi necessário alterar também algumas funções do configurador no que diz respeito à aquisição das referências dos componentes distribuídos. O configurador possui as referências para todos os componentes existentes em seu espaço de endereçamento; isto foi utilizado para garantir a transparência da localização dos componentes na implementação.

A tecnologia Java RMI define um elemento denominado *Registry*, que contém os registros dos componentes que podem ser acessados remotamente. Cada máquina física tem o seu próprio *Registry*, que é acessado pelo configurador no momento de criação de um componente remoto. Após o processamento da especificação da aplicação e a negociação das propriedades, o configurador tem conhecimento da localização dos componentes que fazem parte da aplicação e, através da comunicação entre configuradores, realiza a instanciação dos componentes que se encontram em outro espaço de endereçamento. O configurador responsável pela instanciação do componente deve registrar o mesmo junto ao *registry*, que é acessado posteriormente pelo configurador que trata a especificação da aplicação de forma a recuperar a referência remota do componente em questão.

De posse da referência do componente remoto, o configurador pode passar esta referência aos componentes que necessitam dela, abstraindo a localização dos componentes envolvidos. Com isso, a conexão virtual pode, por exemplo, receber uma

referência da interface *IPortConf* que representa uma porta de comunicação abstraindo a localização desta porta, e caracterizando o funcionamento distribuído do protótipo.

A criação de um componente por parte do configurador foi apresentada na seção 3.4.1. Neste processo, o configurador adquire a referência do componente remoto e retorna esta referência como valor de retorno da chamada de criação. Com as alterações requeridas pela tecnologia Java RMI, a criação de um componente, sob o ponto de vista do componente solicitante, abstrai as questões relacionadas com a localização dos mesmos.

Após a alteração do protótipo para o seu funcionamento com a tecnologia RMI, foram iniciados os testes envolvendo componentes distribuídos. Para isso a aplicação *HelloWorld* [32] foi executada em um ambiente distribuído, onde cada componente é executado em uma máquina física diferente. A escolha da aplicação *HelloWorld* para os primeiros testes distribuídos se deve ao fato desta aplicação já estar funcionando anteriormente, o que facilitou a realização das alterações demandadas pela distribuição com RMI, uma vez que os erros encontrados durante o desenvolvimento do protótipo distribuído estavam relacionados com a distribuição em si, e não com possíveis erros da própria aplicação.

A mudança mais significativa demandada pela distribuição da aplicação *HelloWorld* foi a implementação do critério de escolha para a tecnologia de comunicação a ser empregada pelos canais, pois exigiu uma revisão cuidadosa do funcionamento do configurador no que diz respeito à negociação de propriedades e à instanciação de componentes.

Como citado anteriormente, foram implementados duas tecnologias de comunicação para os canais: em contexto local (*buffer* compartilhado), e em contexto distribuído (*sockets* UDP). O critério para a escolha da tecnologia de comunicação utilizada pelas portas foi baseado na localização dos componentes envolvidos, utilizando o endereço IP da plataforma onde o componente está instanciado. Além disso, também foi utilizado o endereço IP da plataforma para que o *Configurator* decida se o componente deve ser instanciado localmente ou remotamente através da interação com outro *Configurator*.

Para o funcionamento do protótipo de maneira local ou distribuída basta alterar o campo referente à localização dos componentes no arquivo de especificação da aplicação e dos componentes envolvidos.

4.2. Prova de Conceito

Para fins de prova de conceito, o modelo de interconexão foi utilizado e testado com os componentes localizados em uma mesma máquina virtual e em máquinas físicas distintas com diversas topologias de comunicação. Estes testes consistem na instanciação de duas aplicações para o *middleware* AdapTV, uma extensão da aplicação *HelloWorld* apresentado em [32], e em uma adaptação desta aplicação para a transmissão de um fluxo de vídeo.

A extensão do protótipo teve o objetivo de incorporar os conceitos introduzidos pelo modelo de interconexão apresentado na implementação existente. A aplicação *HelloWorld* consiste na comunicação entre dois componentes que realizam a troca de mensagens em diferentes línguas.

A língua utilizada é resultado do processo de negociação realizado pelo configurador antes da instanciação dos componentes. Para esta aplicação, a adaptação consiste na troca da língua utilizada pelos componentes, onde foram realizados testes envolvendo a adaptação pró-ativa e reativa.

Para os testes com a adaptação pró-ativa, foram utilizados os elementos introduzidos pelo modelo de gerenciamento de QoS definido pelo Cosmos. Neste caso, foram definidos monitores para os componentes em questão, utilizando valores randômicos, gerados pelos componentes no momento da consulta de modo a simular a necessidade de adaptação.

Vale salientar que o modelo de recursos virtuais, incluindo as questões relacionadas ao gerenciamento e alocação de recursos, bem como modelo de gerenciamento de QoS e o funcionamento dos monitores estão fora do escopo deste trabalho.

Os testes envolvendo a adaptação reativa foram realizados através de interações disparadas por meio de uma interface gráfica simplificada que é apresentada pela Figura 44. Esta interface é exibida pelo componente receptor, e apresenta em uma janela, os dados recebidos pelo componente, permitindo ao usuário a alteração da língua corrente. O usuário, utilizando esta interface gráfica, seleciona uma nova língua e dispara o processo de adaptação que é controlado pelo *Configurator*.

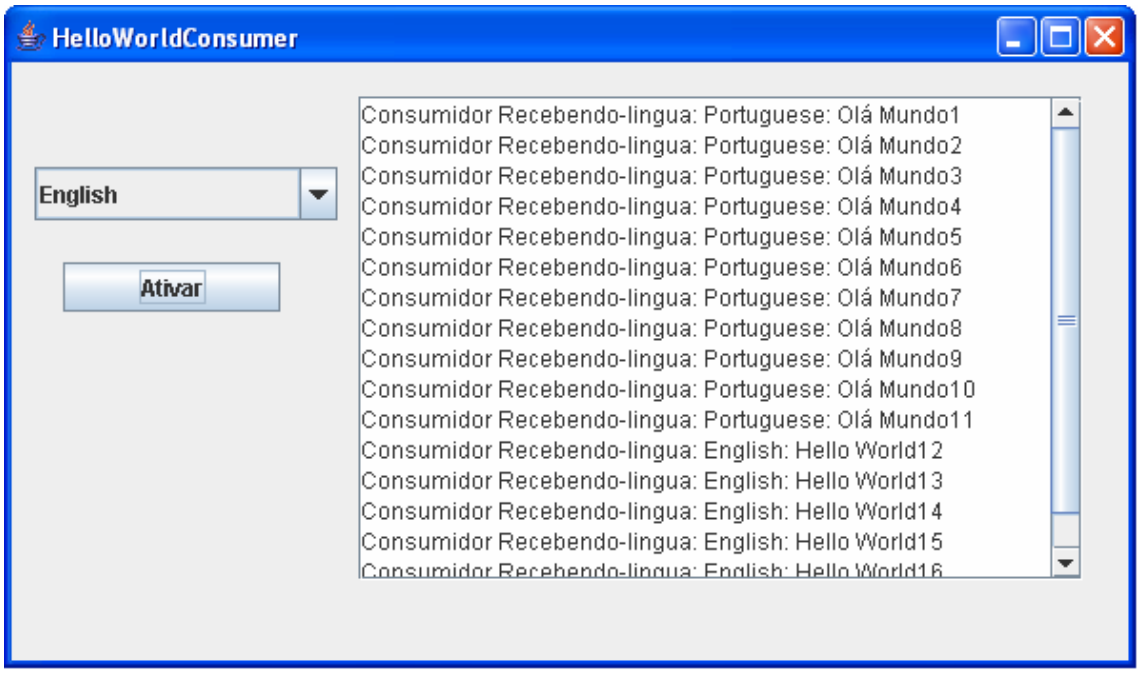

Figura 44. Interface Gráfica do Componente Consumidor da Aplicação HelloWorld.

Os testes iniciais foram realizados em um ambiente local, com os componentes envolvidos executando em uma mesma máquina virtual. Em seguida, foi realizada uma alteração no protótipo, de forma a suportar a execução distribuída incorporando a tecnologia Java RMI.

Estes testes foram realizados utilizando duas, e em seguida três máquinas físicas distintas envolvendo diversas topologias para as aplicações testadas como 1x1, 1x2, 2x1, 3x1 e 3x2.

4.2.1. Aplicação de transmissão de vídeo

Além da aplicação *HelloWorld*, foram realizados testes com transmissão de vídeo, onde os componentes envolvidos realizam a troca de um fluxo de vídeo em formato MPEG1. Esta aplicação possui a mesma arquitetura da aplicação *HelloWorld*, formada por um componente transmissor e um componente receptor, diferenciando no tipo de dado transmitido e manipulados pelos componentes.

A implementação, com sua arquitetura apresentada na Figura 45, consiste em um componente transmissor (*Sender*) e um componente receptor (*Receiver*). O componente transmissor tem acesso a três arquivos distintos com o mesmo vídeo codificado com qualidades diferentes. Inicialmente, o vídeo é transmitido com a qualidade negociada pelo configurador, e durante a execução, o usuário, através da interface gráfica citada, inicia o processo de adaptação escolhendo uma nova qualidade para o vídeo.

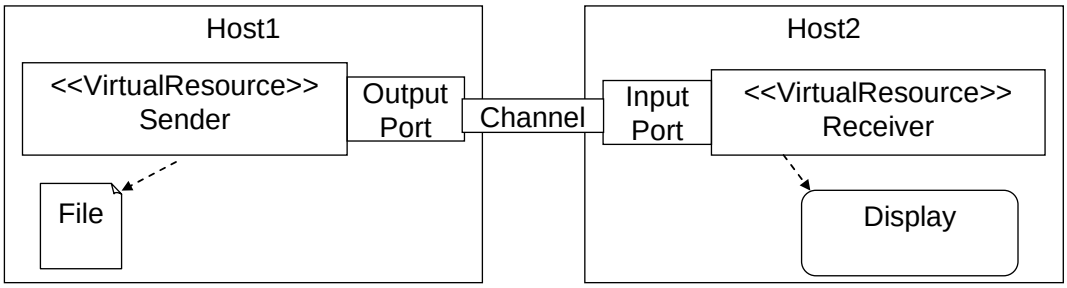

Figura 45. Arquitetura da Aplicação com Vídeo.

Neste caso, a adaptação consiste na troca do arquivo fonte e em sua sincronização, de forma que o vídeo seja exibido sem interrupção para o usuário. Esta sincronização consiste em um ajuste de tempo realizado pelo componente transmissor no momento da abertura do novo arquivo.

A Figura 46 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>
```

Figura 46. Descrição XML Componente *VideoFlowProducer*.

A Figura 47 mostra a especificação da configuração 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. Nesta aplicação, o modelo de interconexão é usado para a realização de adaptações envolvendo mudança de propriedades do fluxo (*framerate*) durante a operação do sistema.

```
<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>
```

Figura 47. Descrição XML da configuração da aplicação de vídeo.

Conforme a descrição da Figura 47, 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*.

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.

Para a exibição desta aplicação utilizou-se a mesma interface da aplicação *HelloWorld*, realizando apenas alterações para a exibição de um fluxo de vídeo nesta interface. A Figura 48 apresenta duas telas extraídas do fluxo correspondente à execução da aplicação no momento da transição das propriedades do fluxo, onde se percebe a diferença de qualidades entre o vídeo com a qualidade negociada (a), e o vídeo após a adaptação com melhor qualidade (b).

Figura 48. Vídeo Antes (a) e Depois (b) da Adaptação

Para a manipulação do fluxo de vídeo, foi utilizado o *Java Media Framework* (JMF) [35] desenvolvido pela *Sun Microsystems* com o objetivo de dar suporte à manipulação de fluxos multimídia em aplicações Java. O protótipo implementado utiliza o JMF para o tratamento dos aspectos relacionados ao formato e à exibição do fluxo. Com isso, as questões relacionadas com a interpretação dos dados, tanto do lado transmissor, quanto do lado do receptor, são abstraídas.

O JMF oferece uma solução para a transmissão de fluxos multimídia, fornecendo a possibilidade de utilização de diversos protocolos para a transmissão via rede. Para a implementação do protótipo, foram utilizadas as capacidades de tratamento

do fluxo relacionadas com os formatos oferecidas pelo JMF, e, para a transmissão/transporte dos dados do fluxo entre o transmissor e o receptor, desenvolveu-se uma solução própria envolvendo o conceito de portas, definido na arquitetura de forma desacoplada do JMF.

Para isso, foi realizada uma modelagem envolvendo os componentes do JMF envolvidos em uma transmissão de um fluxo de vídeo com o objetivo de elucidar as responsabilidades de cada componente. Com a definição dos papéis dos componentes do modelo, iniciou-se um processo de implementação de componentes JMF customizados que foram agregados a componentes Cosmos, de forma que a comunicação entre os componentes se desse através das portas de comunicação.

Do lado do transmissor, foi implementado um *DataSink* customizado, responsável por receber o fluxo de um componente JMF e encaminhá-lo para um componente Cosmos. Do lado do receptor, foi utilizado um *DataSource* customizado, que recebe os dados da porta e realiza o tratamento necessário para que os dados sejam entregues a um componente JMF, neste caso um *player* que realiza a interpretação dos dados e a exibição do vídeo.

A Figura 49 apresenta as classes envolvidas na implementação do exemplo de transmissão de vídeo. As classes *SourceComponent* e *SinkComponent* representam os componentes transmissor e receptor, respectivamente. Os dois componentes utilizam as classes definidas pelo JMF para o tratamento do fluxo. O componente transmissor incorpora um *processor*, que é o elemento responsável pelo processamento e controle de mídias. O *processor* recebe os dados através de um componente *DataSource*, que por sua vez utiliza a classe *File* para representar o arquivo que contém o vídeo a ser transmitido. O *DataSource* é um gerenciador de protocolo de uma determinada mídia, responsável por gerenciar o ciclo de vida de uma fonte de mídia provendo um protocolo simples de comunicação.

A saída de um *processor* pode ser utilizada por qualquer um dos elementos do JMF; devido a isso, ela é representada através de um *DataSource*. Este *DataSource* é utilizado pelo elemento que efetivamente realiza a transmissão, ou salva o conteúdo em um arquivo, o *DataSink*. A classe *DataSourceHandler* implementa a interface *DataSink*, sendo responsável por encaminhar os bytes que formam o fluxo para a porta de saída.

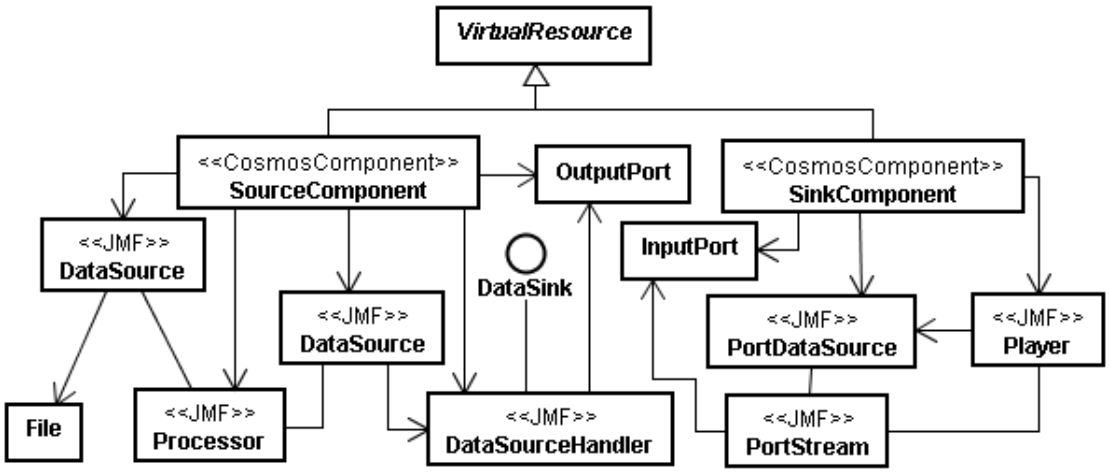

Figura 49. Classes JMF acessadas por Componentes Cosmos.

O componente receptor, *SinkComponent*, é o componente que recebe o fluxo de vídeo e exibe o mesmo para o usuário. Ele incorpora um objeto da classe *Player* do JMF, que efetivamente realiza o tratamento dos dados recebidos e realiza a exibição do vídeo em uma janela. Para a recepção dos dados por parte do *player* é necessário fornecer a ele um *DataSource*. Este *DataSource* disponibiliza um *Stream*, que é utilizado pelo player para a aquisição dos dados. No protótipo, implementou-se um *DataSource* e um *Stream* customizado, *PortDataSource* e *PortStream*, de forma que o fluxo a ser recebido pelo *player* seja proveniente da porta de comunicação.

Inicialmente, foram realizados testes envolvendo a adaptação reativa em um ambiente local. O *SinkComponent* apresenta em sua interface gráfica um menu de opções onde o usuário realiza a seleção de um entre os possíveis valores para a propriedade afetada. Em seguida, incorporou-se neste exemplo a utilização de monitores de QoS, que recebe valores gerados randomicamente pelos componentes recursos virtuais para provocar necessidades de adaptação.

Após os testes com adaptação reativa e pró-ativa em um ambiente local, iniciaram-se os testes com a distribuição da aplicação de vídeo. Como citado anteriormente, bastou alterar a especificação da aplicação e a instanciação dos componentes de modo que eles refletissem a distribuição da aplicação.

Além da distribuição dos componentes em máquinas distintas, foram realizados testes envolvendo diversas topologias de comunicação, variando-se o número de componentes envolvidos. Para a aplicação de vídeo, foram realizados testes envolvendo as seguintes topologias: 1x1, 1x2, e 1x3.

Os testes realizados com a transmissão de vídeo em um ambiente distribuído contemplaram o processo de adaptação reativa e, apesar de preliminares, se mostraram bastante promissores. Para a realização de testes distribuídos envolvendo a adaptação pró-ativa, é necessário realizar a alteração nos componentes do modelo de gerenciamento de QoS para que os mesmos incorporem os requisitos definidos pela tecnologia RMI.

4.3. Considerações Finais

O protótipo do *middleware* AdapTV foi concebido de forma a avaliar os conceitos e a arquitetura do *framework* Cosmos, servindo como prova de conceito para o *framework*. Seu desenvolvimento foi concentrado nos aspectos relacionados com a negociação de propriedades e adaptação com uma abordagem estática.

O AdapTV foi estendido para incorporar o modelo de interconexão apresentado. Esta extensão foi desenvolvida como prova de conceito para o modelo de interconexão. Neste protótipo foram realizados testes envolvendo a interconexão de componentes e a adaptação dinâmica entre estes componentes. Para isso, explorou-se uma aplicação exemplo definida no protótipo inicial, de maneira a incorporar os conceitos introduzidos pelo modelo de interconexão, para em seguida definir uma nova aplicação envolvendo a transmissão de fluxos multimídia.

Esta implementação demonstra o potencial da arquitetura de interconexão proposta, contribuindo para conceber uma visão integrada de todo o sistema. Adicionalmente, ela revela alguns aspectos que devem ser considerados de forma a refinar os outros modelos definidos pelo *framework* Cosmos, como por exemplo, o modelo de recursos virtuais e o modelo de gerenciamento de QoS.

Os testes realizados envolveram inicialmente a comunicação entre um componente transmissor e um componente receptor, caracterizando uma conexão *unicast*, tanto em um ambiente local quanto em um ambiente distribuído. Em seguida, foram realizadas instanciações de aplicações envolvendo diversas topologias de comunicação, com diversos cenários na distribuição dos componentes envolvidos.

Estes testes contemplaram o processo de adaptação dinâmica, em todos os cenários testados, envolvendo as duas aplicações definidas. Do ponto de vista do usuário, percebe-se a mudança do parâmetro operacional dos componentes envolvidos através da troca da qualidade do fluxo de vídeo exibido, no caso da aplicação que

envolve a transmissão de vídeo entre dois componentes, e da mudança da língua utilizada para a transmissão das mensagens no caso da aplicação *HelloWorld*.

Entretanto, para resultados mais precisos, faz-se necessário a realização de testes mais rigorosos, envolvendo outras tecnologias de comunicação, e outras soluções para o tratamento e exibição de fluxos de mídia além do JMF, além de alterações nos outros modelos envolvidos.

5. Trabalhos Relacionados

A existência de diversas propostas de *middlewares* adaptativos e de seus respectivos modelos de comunicação voltados para ambientes multimídia evidencia a necessidade da definição de mecanismos de comunicação específicos para este tipo de ambiente. O *framework* Cosmos é um *framework* genérico baseado em componentes para a camada de *middleware* de uma variedade de sistemas multimídia distribuídos, definindo um conjunto de conceitos com o objetivo de oferecer suporte a reconfiguração dinâmica de componentes.

Com isso, o Cosmos também necessita de mecanismos de comunicação específicos para a transmissão de fluxos de informações. Contudo, as propostas existentes não podem ser incorporadas ao Cosmos fazendo com que seja necessária a definição de um mecanismo de comunicação adequado para a transmissão de fluxos de informações.

Este capítulo apresenta uma discussão sobre algumas das propostas de mecanismos de comunicação para a transmissão de fluxos de informações existentes, e que não podem ser incorporadas ao *framework* Cosmos, destacando suas principais características e realizando algumas comparações da proposta apresentada com estas soluções existentes, evidenciando suas semelhanças e diferenças.

5.1. PREMO

O *framework* PREMO (*Presentation Environment for Multimedia Objects)* [16], proposto pela ISO, foi baseado no modelo RM-ODP. Ele 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. Seus principais elementos são as abstrações para dispositivos (*VirtualDevice*s), portas (*port*) e conexões virtuais (*Virtual Connections*). Estas entidades suportam conexões *ponto-ponto* e *ponto-multiponto*.

Um recurso virtual representa uma unidade de processamento, podendo ser uma entidade de software ou uma entidade de hardware. As portas são utilizadas para realizar a ligação entre recursos virtuais. A porta suporta diversos formatos de mídia associados que podem ser consultados ou configurados. A conexão virtual abstrai questões relacionadas ao gerenciamento e transferência de dados entre os dispositivos. Ela não realiza a transferência de dados entre os dispositivos. Ela é responsável por separar a conexão em diferentes elementos.

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. Entretanto, o modelo para conexão virtual definido no PREMO não permite que a aplicação se envolva na definição de propriedades do processamento e transporte do fluxo, o que inviabiliza a realização de adaptação e não permite sua incorporação ao *framework* Cosmos.

5.2. A/V Streams

Os conceitos do PREMO influenciaram outras propostas como o *framework* A/V *Streams* [36] da OMG. O A/V *Streams* é um padrão aberto para a arquitetura CORBA [37]. Sua especificação define componentes e serviços com uma abordagem arquitetural para implementação e controle de aplicações envolvendo fluxos multimídia. Todas as operações de controle são realizadas através do ORB, enquanto que os dados de um fluxo multimídia são transportados por protocolos adequados para este fim (TCP, UDP, RTP, etc).

O A/V *Streams* define um conjunto de módulos, interfaces e semânticas para a transmissão de fluxos de informação. Os recursos são representados através do conceito de *flow endpoints* (pontos terminais de fluxo), e podem ser especializados para se comportarem como produtores de fluxo (*FlowProducer*), ou consumidores de fluxo (*FlowConsumer*). Os pontos terminais de fluxo utilizam um elemento denominado adaptador de fluxo (*Stream Adapter*) para a transmissão de dados. O adaptador de fluxo tem um papel semelhante ao das portas do modelo de interconexão apresentados.

O A/V *Streams* suporta ligações ponto-a-ponto e ponto-multiponto através de conexões (*flow connection*), envolvendo um produtor e um ou mais consumidores. A conexão é realizada através da abordagem de *binding* explícito, ficando a cargo da aplicação o mapeamento de parâmetros de QoS em tipos oferecidos pela plataforma.

O A/V *Streams* é um *framework* baseado no PREMO, aderente a arquitetura CORBA. Devido a isso, ele incorpora as limitações existentes no PREMO, o que impossibilita sua incorporação ao *framework* Cosmos. Assim como o PREMO, o A/V *Streams* oferece suporte a reconfiguração, possuindo um elevado nível de complexidade.

Além disso, as portas do modelo de interconexão apresentado são tratadas como componentes, o que permite a sua configuração e reconfiguração dinâmica através de suas interfaces de propriedades.

5.3. NMM

O projeto NMM (*Network-Integrated Multimedia Middleware*) [23] provê uma arquitetura baseada em objetos para a construção de aplicações multimídia distribuídas em ambiente GNU/Linux. O NMM foi desenvolvido utilizando a linguagem C++, e hoje se tornou um projeto de software livre sendo distribuído sob as licenças GPL [38] e LGPL [39]. Um sistema NMM consiste de uma coleção de nós (*nodes*), que são entidades de software independentes e compõem as unidades básicas do sistema. Os nós constituem um grafo, formando um *pipeline* onde cada nó realiza uma função específica no fluxo multimídia.

Os nós (*nodes*) são os elementos funcionais do NMM, representando dispositivos de hardware e software. Os nós se conectam através dos *jacks*, que representam os pontos de entrada e saída dos nós. Os *buffers* contêm os dados que são transportados através dos *jacks* de um nó para o outro, sendo gerenciados por um gerente (*buffermanager*). Um formato (*format*) é uma descrição da estrutura e função dos dados que compõem um fluxo multimídia. Os formatos determinam os tipos de dados de um fluxo e seus parâmetros. Os nós são conectados formando um grafo, onde o formato de dois nós que se comunicam devem ser iguais.

Toda comunicação no NMM utiliza um sistema de mensagem unificada que é composta por dois tipos de objetos: *Buffers*, utilizado para a transmissão de dados multimídia, que podem estar em uma fila para ser processado; e eventos compostos (*Composite Events*), que são utilizados para controlar o comportamento de um nó. As mensagens de eventos são enviadas de duas maneiras: *in-stream* e *out-of-band*. Nos eventos *in-stream*, as mensagens são encaminhadas da mesma maneira que em *buffers*. Nos eventos *out-of-band*, as mensagens são enviadas se comunicando diretamente com

os nós instanciados através de chamadas remotas. Um evento composto consiste de um conjunto de eventos simples (ou comandos) ordenados.

A Figura 50 mostra a anatomia de um nó NMM, com seus *jacks* de entrada e saída. Através do *jack* de entrada o nó recebe *buffers* ou eventos *in-stream*, que são tratados de maneiras distintas. Ao mesmo tempo, o nó pode receber eventos *out-of-band* da aplicação ou diretamente de outros nós, que correspondem a eventos de controle. Através de seu *jack* de saída, ele envia *buffers* ou eventos que serão tratados pelo nó seguinte ou encaminhado para o próximo nó da cadeia.

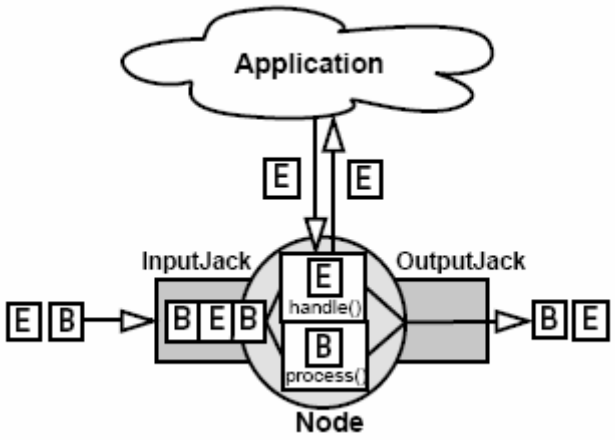

Figura 50. Anatomia de um Nó NMM.

Os *jacks* são responsáveis por transportar os dados dos *buffers* de um nó para o outro, representando a entrada e saída de um nó. Os nós são criados com seus formatos de entrada e de saída definidos, e, posteriormente, os *jacks* associados são criados e conectados. Diversos *jacks* de saída podem ser combinados em um *container* chamado *JackGroup*, permitindo o envio de dados para diversos receptores de forma transparente para o emissor. Através do uso dos *jackGroups*, o NMM oferece suporte à comunicação um-para-muitos (1xN) com o número de receptores podendo variar dinamicamente.

O *middleware* NMM define uma arquitetura distribuída baseada em objetos para a construção de aplicações multimídia explorando os conceitos definidos pelo RM-ODP [24]. Entretanto, sua implementação está voltada para o contexto de ambientes GNU/Linux. Diferentemente, o modelo de interconexão proposto neste trabalho oferece uma arquitetura genérica baseada em componentes que pode ser incorporada a diversos modelos existentes.

Através do uso de *jacks* o NMM explora o conceito de portas de comunicação do PREMO [16] e do A/V Streams [20], possibilitando conexões 1xN de forma transparente para os elementos participantes. Os *jacks* funcionam de maneira análoga às portas do modelo de interconexão proposto, porém, nesse último, diversas topologias de interconexão são suportadas (1x1, 1xN, Nx1 e NxN), permitindo a variação do número de participantes de acordo com as necessidades da aplicação. Além disso, por se tratarem de componentes, as portas do modelo de interconexão proposto podem ser configuradas através de uma linguagem de especificação e reconfiguradas automaticamente mediante processo de negociação em tempo de execução. Os *jacks* do NMM só são criados após a definição do formato a ser utilizado pelos nós, não suportando nenhum tipo de configuração.

Com a definição dos formatos suportados por um nó, o NMM suporta negociação antes da criação de um grafo, tendo como indicação de trabalho futuro a inserção de elementos adaptadores quando necessário. Isto pode acarretar a perda de controle, por parte do desenvolvedor, dos elementos que compõem o grafo. Questões relacionadas com a adaptação e reconfiguração dinâmica no NMM são indicadas como trabalhos futuros, não tendo até o presente momento nenhum suporte.

5.4. DIRECTSHOW

O *framework DirectShow* [40] da Microsoft facilita a construção de aplicações multimídia em ambientes *Microsoft* através da tecnologia de componentes COM. Ele se integra facilmente ao ambiente Windows e possui dois tipos de objetos: filtros, as unidades atômicas do *framework*; e grafos de filtros, uma coleção de filtros interligados. Os filtros, por sua vez, possuem pinos, que funcionam como pontos de conexão, e são utilizados para enviar e receber dados.

O grafo de filtros do *DirectShow* é uma estrutura estática, não oferecendo suporte para reconfiguração dinâmica. Uma vez definido um grafo, este não pode mais ser alterado. Eventuais mudanças no estado do ambiente que impliquem na alteração do grafo de filtros exigem a destruição do grafo existente, com a desalocação dos recursos e a definição de um novo grafo com uma nova reserva e alocação de recursos, de modo a refletir o novo estado do ambiente.

A conexão entre os filtros se dá através dos pinos. Para isto é preciso que haja concordância entre o tipo de fluxo que eles irão trocar e o mecanismo de transporte que

será utilizado para troca de dados. Existem dois mecanismos de transporte utilizados pelos filtros do *DirectShow*, ambos locais, se diferenciando na maneira e no espaço em que a memória é alocada: na memória principal do computador, ou na memória de algum dispositivo de hardware conectado ao computador.

O processo de desenvolvimento de um filtro *DirectShow* se resume a desenvolver um objeto COM, com as interfaces características de um filtro *DirectShow*. Esta modularidade se estende para os grafos de filtros, isto é, assim como os detalhes de um filtro ficam escondidos para o programador, as partes internas de um grafo também podem ser escondidas do programador.

É possível criar um grafo bastante complexo através de uma funcionalidade chamada *Intelligent Connect*. Esta funcionalidade examina os filtros no grafo e determina a melhor maneira de conectá-los, adicionando os filtros de conversão necessários para o funcionamento. Apesar das vantagens inerentes oferecidas por esta funcionalidade, o programador não tem conhecimento de quais filtros exatamente fazem parte do grafo. Devido a isso, caso exista dois filtros que desempenham a mesma função, o programador não saberá qual está sendo utilizado. O programador tem a liberdade de definir quais filtros ele deseja utilizar, mas, para isso, tem-se um nível de complexidade maior para definição da aplicação.

Como mencionado, o processo de conexão é realizado pelos pinos, onde são tratadas negociações entre filtros e acordos envolvendo os tipos de dados trocados entre eles, bem como os mecanismos de transporte utilizados para a comunicação. O processo de negociação é realizado a partir de uma lista de tipos que cada filtro publica; nesta lista estão os tipos de dados que eles podem receber ou enviar.

O *framework DirectShow* apresenta uma solução simples, mas restrita a ambientes *Microsoft*. Ao utilizar o modelo de componentes COM, ele permite a rápida construção de uma aplicação multimídia e, explorando o conceito de grafos de filtros, o programador não precisa conhecer todos os membros do grafo. Através da funcionalidade chamada *Intelligent Connect*, filtros intermediários são adicionados ao grafo garantindo a compatibilidade de dados entre os extremos deste grafo. O programador pode se assim desejar, definir todos os membros do grafo, mas este é um processo mais trabalhoso e complexo.

O *DirectShow* é restrito a aplicações locais não suportando nenhum tipo de adaptação ou reconfiguração e, no caso de aplicações distribuídas, o programador tem que explicitamente configurar duas aplicações em máquinas distintas e certificar se elas

estão configuradas para enviar/receber do dispositivo correto, além de configurar o protocolo de comunicação a ser utilizado. Os pinos do *DirectShow* suportam somente comunicação 1x1. Para topologias 1xN, o desenvolvedor precisa adicionar um filtro específico com diversos pinos de saída, um para cada receptor.

As portas do modelo de interconexão proposto funcionam de maneira análoga aos pinos do *DirectShow*, podendo ser usadas para transmitir fluxos de dados. Entretanto, no modelo de interconexão proposto existe uma checagem prévia de compatibilidade das portas no estabelecimento da comunicação, antes da instanciação dos componentes. Além disso, as portas são tratadas como componentes podendo ser configuradas e reconfiguradas em tempo de execução.

A definição do mecanismo de transporte a ser utilizado para interligar as portas é realizada mediante processo de negociação, suportando diversos tipos de comunicação local e distribuída, com a possibilidade de adição de um novo protocolo de comunicação, por parte do desenvolvedor. Além disso, o modelo de interconexão proposto neste trabalho suporta diversas topologias para a interligação dos componentes que fazem parte de uma aplicação de maneira transparente para os mesmos, oferecendo suporte a reconfiguração dinâmica.

5.5. InfoPIPE

O Infopipe [20] é uma plataforma de *middleware* para aplicações de fluxo de informações baseada no modelo arquitetural produtor-consumidor. Ele trata a tarefa de construir aplicações distribuídas orientadas a fluxo através da abordagem de abstrações para os mecanismos de transporte. Para isso, ele modela os componentes de um *pipeline* de maneira análoga a um sistema de fornecimento de água. Ele define componentes básicos como tubos (*pipes)*, filtros (*filters*), *buffers* e bombas (*pumps*), suportando a composição de *pipelines* formados por estes componentes básicos. Estes componentes se comunicam através de portas.

Os componentes mais comuns possuem uma porta de entrada e uma porta de saída. Estes componentes podem filtrar ou transformar os dados de um fluxo de informação. Os *buffers* oferecem um espaço de armazenamento temporário que visam eliminar as flutuações de taxas de transmissão, condições causadoras de *jitter*. As bombas são utilizadas para manter o fluxo de informação.

Cada porta tem uma polaridade, que define o elemento ativo na ligação entre duas portas. Uma porta de saída positiva empurra itens, enquanto que os itens são puxados de uma porta de saída negativa. De maneira análoga, uma porta de entrada positiva puxa itens, enquanto que os itens são empurrados para uma porta negativa. As bombas possuem duas portas positivas, os *buffers* duas portas negativas, e os filtros e transformadores possuem duas portas de polaridade opostas. Fontes e destinos de dados possuem somente uma porta, podendo ser positiva ou negativa. Duas portas conectadas precisam ser de polaridade invertida, O Infopipe oferece mecanismo para a checagem de consistência nestas ligações.

Componentes mais complexos possuem mais portas. Como exemplo, temos os *tees* para a separação e a junção de fluxos. Protocolos de transporte podem ser integrados ao *middleware* encapsulando os mesmos como *netpipes*. Estes *netpipes* suportam o conceito de fluxos de dados, permitindo a manipulação e o gerenciamento de propriedades de baixo nível como largura de banda e latência. Filtros de composição e serialização podem ser utilizados para realizar o mapeamento dos dados do fluxo tanto para o formato de alto-nível da camada de nível superior, quanto para a camada de nível mais baixo. Estes componentes encapsulam também o mapeamento de QoS, realizando a conversão entre as propriedades de um *netpipe* e as propriedades específicas do fluxo.

Para construir um Infopipe, o desenvolvedor da aplicação precisa combinar os filtros, *buffers*, bombas e *netpipes*. Ao combinar estes componentes é necessário verificar a compatibilidade dos fluxos suportados e avaliar as características do Infopipe composto. Cada Infopipe, básico ou composto, possui uma especificação de tipo que descreve o fluxo que ele suporta. Esta especificação, denominada *typespecs*, provê informações sobre os formatos suportados, as propriedades de interação como a polaridade das portas, e a faixa de parâmetros de QoS que podem ser tratados. A plataforma oferece mecanismos para a realização da checagem de tipos entre os componentes.

A Figura 51 apresenta um *pipeline* de exemplo de uma aplicação de vídeo, com uma câmera produzindo dados comprimidos para exibição em uma tela. No lado da origem, os quadros são produzidos por uma câmera, representada pelo objeto *source*, e empurrados por uma bomba (*pump*) através de um filtro para um *NetPipe* que encapsula os dados através de um protocolo de transporte.

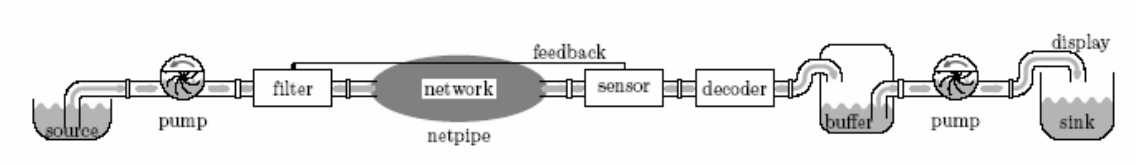

Figura 51. Pipeline de Exemplo.

O filtro realiza o descarte de quadros quando a rede está congestionada. Este descarte é controlado por um mecanismo de *feedback* utilizando um sensor no lado do destino. Este *feedback* indica qual dado descartar ao invés de sofrer o descarte aleatório pela rede. Após a decodificação dos quadros, eles são armazenados em um *buffer* com o objetivo de reduzir o *jitter*. Os dados são removidos do *buffer* por uma bomba que encaminha os mesmos para a tela.

A plataforma Infopipe oferece um modelo de componentes para a transmissão de fluxos de informações realizando uma analogia a um sistema de fornecimento de água. A plataforma utiliza a abordagem de representar os elementos que compõem os mecanismos de comunicação utilizando diversas abstrações para os mecanismos de comunicação, como fontes, *buffers*, bombas ou filtros, facilitando a definição de *bindings* explícitos. Uma vez que a plataforma não oferece mecanismos para a configuração automática dos componentes que realizam a ligação entre um componente produtor e um componente consumidor, o desenvolvedor precisa especificar estes componentes explicitamente, adicionando um nível de complexidade no desenvolvimento de uma aplicação.

Os componentes do Infopipe se comunicam através de portas. Estas portas funcionam de maneira análoga às portas do modelo de interconexão proposto neste trabalho, como elementos de entrada ou saída de dados, tendo como diferencial a definição de polaridade para as mesmas, o que adiciona um nível de complexidade ao desenvolvedor, que precisa garantir que as portas conectadas sejam de polaridades diferentes. Além disso, as portas do modelo de interconexão são tratadas como componentes do *framework* Cosmos, podendo ser configuradas e reconfiguradas dinamicamente em tempo de execução, oferecendo suporte para reconfigurações dinâmicas.

O Infopipe define componentes com diversas portas. Estes componentes podem realizar algum tratamento no fluxo, ou simplesmente funcionar como um mecanismo para possibilitar o suporte a diferentes topologias de conexão. O desenvolvedor precisa explicitamente selecionar estes componentes divisores de forma a definir topologias

distintas de conexão. No modelo de interconexão, as diferentes topologias de interconexão são construídas de maneira transparente ao desenvolvedor da aplicação, podendo ser alteradas em tempo de execução através de reconfiguração dinâmica.

5.6. Considerações Finais

As plataformas para sistemas multimídia apresentadas possuem algumas limitações na definição de mecanismos de comunicação para este tipo de ambiente, o que inviabiliza sua integração ao *framework* Cosmos. O modelo de interconexão apresentado é baseado na tecnologia de componentes, oferecendo uma solução genérica e flexível, e de fácil integração ao *framework* Cosmos.

O *framework* PREMO [16] possui um elevado nível de complexidade, inviabilizando sua aplicação no desenvolvimento de sistemas multimídia atuais. Entretanto, vale destacar que os conceitos do PREMO influenciaram outras propostas como o *framework* A/V Streams [36], e o próprio Cosmos [4].

A abordagem orientada a objetos do NMM, e seu foco para ambientes de execução Linux restringe as possibilidades de utilização do *middleware*. Com a abordagem da tecnologia de componentes, o modelo de interconexão pode ser facilmente incorporado a outros modelos existentes. O NMM é baseado no RM-ODP, utilizando o conceito de portas de comunicação através da definição de *jacks*. Entretanto estes elementos não são passíveis de configuração, dificultando a realização de ajustes em tempo de execução. A abordagem do NMM permite somente conexões ponto-multiponto, com uma configuração estática anterior à instanciação dos componentes. O modelo de interconexão suporta diversas topologias de comunicação permitindo seu ajuste em tempo de execução.

O *DirectShow* utiliza uma abordagem que se aproveita do modelo de desenvolvimento baseado em componentes através do modelo de componentes COM da Microsoft. Entretanto, este modelo é restrito a ambientes *Windows*, e dependente de outros componentes do ambiente. Além disso, o *Directshow* é restrito a aplicações locais suportando somente comunicação ponto-ponto. Para a definição de outras topologias é necessária a inclusão de elementos divisores com diversos pontos de saída. O *Directshow* define mecanismos para a negociação entre os componentes, mas não suporta nenhum tipo de alteração em tempo de execução.

A plataforma Infopipe utiliza a abordagem de expor ao desenvolvedor os detalhes dos mecanismos de comunicação, favorecendo a definição de *bindings* explícitos. Entretanto, esta abordagem torna difícil a definição de aplicações. O Infopipe explora o conceito de porta de comunicação para a ligação entre seus elementos, adicionando um nível de complexidade ao definir polaridades para as portas, fazendo com que além das restrições de uma porta de saída estar conectada a uma porta de entrada, estes elementos têm que possuir polaridades diferentes. Além disso, o suporte a diversas topologias é conseguido através da inserção de elementos divisores de fluxo, como no *Directshow*.

A principal diferença entre as soluções apresentadas e o modelo de interconexão apresentado é o fato do modelo de interconexão ser baseado na tecnologia de desenvolvimento baseado em componentes enquanto que os trabalhos existentes são baseados na tecnologia de orientação a objetos. Além do suporte a reflexão computacional, não oferecida pelas soluções para fluxos de informação apresentadas.

O modelo explora a tecnologia de componentes ao definir as portas de comunicação como componentes que podem ser configurados e reconfigurados em tempo de execução. O conceito de portas de comunicação, utilizado pelas outras plataformas apresentadas, é simplificado sob a visão do usuário que só precisa conhecer as interfaces operacionais das portas. A responsabilidade pela configuração dos elementos constituintes do mecanismo de comunicação é atribuída ao *middleware*, que o faz através das interfaces de configuração.

Os conceitos de conexão virtual e canal de comunicação permitem o suporte a diferentes topologias de conexão, utilizando diversas tecnologias de comunicação de maneira simultânea e transparente. Além disso, estes conceitos permitem a definição de uma solução simplificada para o suporte a reconfiguração dinâmica.

6. Conclusão

O presente trabalho apresentou um modelo de interconexão de componentes para ambientes distribuídos multimídia. As atuais tecnologias de *middleware* apresentam muitas soluções interessantes para conectividade. Em relação a estas soluções, este trabalho apresenta uma abordagem diferente, relacionada à flexibilidade e simplicidade para interconexão de componentes. O modelo explora o conceito de portas de comunicação para abstrair os detalhes relacionados a protocolos de comunicação. Este modelo foi definido sendo incorporado ao *framework* Cosmos, utilizando o conceito de conexão virtual definido pelo modelo PREMO, e simplificando a integração, uso e gerenciamento dos componentes envolvidos no modelo.

O conceito de portas, vastamente explorado pela engenharia de software, é utilizado no modelo para dar suporte a interações envolvendo fluxos contínuos entre componentes. Cada componente pode possuir várias portas, cada uma com tipo e propriedades diferentes. As portas também são tratadas como componentes, assim como todos os outros elementos do modelo. O modelo define um conjunto de interfaces com operações funcionais e de configuração, contemplando diferentes semânticas.

Componentes locais ou remotos são interconectados através das portas de comunicação, que por sua vez são gerenciadas pela conexão virtual (*VirtualConnection*). O componente interage com o ambiente através das portas de comunicação, bastando que ele conheça somente as interfaces operacionais definidas. Os detalhes de configuração e gerenciamento são abstraídos para os componentes da aplicação, uma vez que são de responsabilidade do *middleware*.

O modelo suporta diversas topologias de interconexão, envolvendo de maneira simultânea, diferentes tecnologias de comunicação. Isto é conseguido através da definição dos canais de comunicação, que realizam ligações entre portas de saída e portas de entrada. Além disso, o modelo foi definido de forma a suportar a adaptação dinâmica de componentes utilizando técnicas como a clonagem e uso simultâneo e temporário de canais de comunicação.

O modelo apresenta uma arquitetura de interconexão genérica, o que possibilita a integração de uma variedade de tipos de componentes em ambientes heterogêneos e distribuídos. Através do uso do conceito de propriedades explorado pelo *framework* Cosmos, o modelo oferece uma API simples e bastante flexível.

Para a avaliação dos conceitos introduzidos pelo modelo de interconexão foi realizado uma extensão no protótipo de um *middleware* adaptativo para sistemas de TVDI, denominado AdapTV. O AdapTV é uma instância do *framework* Cosmos para sistemas de TVDI, com o objetivo de servir como prova de conceito para os elementos do *framework*. Esta validação envolveu a definição de duas aplicações-exemplo.

Estas aplicações fizeram uso das funcionalidades definidas pelo *framework* Cosmos e pelo modelo de interconexão proposto, realizando testes envolvendo a adaptação dinâmica dos componentes da aplicação em um contexto local e distribuído envolvendo diversas topologias de interconexão. A adaptação dinâmica dos componentes pode ser realizada de duas maneiras, através da reconfiguração de parâmetros operacionais dos componentes ou através da troca de componentes. O *framework* Cosmos oferece o suporte para a realização da adaptação interna em componentes, fazendo com que o modelo de interconexão e os exemplos implementados tivessem como foco a definição de conceitos e técnicas para o suporte a este tipo de adaptação.

O protótipo conta com aproximadamente 10.000 linhas de códigos em 100 classes Java distribuídas em interfaces, classes abstratas e classes concretas. Dentre os conceitos implementados e integrados ao protótipo temos o modelo de meta-dados e um *parser* XML, ambos definidos em [29]. Temos ainda o modelo de gerenciamento de QoS [31], que funciona em conjunto com o modelo de interconexão apresentado pelo presente trabalho.

Para a realização de testes em um ambiente distribuído foi utilizada tecnologia de chamada de métodos remotos através da tecnologia Java RMI. Para a distribuição do modelo, as interfaces definidas para os componentes que terão algum tipo de acesso remoto foram alteradas de forma a incorporar as mudanças requeridas pela tecnologia RMI. Além disso, foram necessárias mudanças no funcionamento do componente configurador definido pelo *framework* Cosmos. Estas mudanças dizem respeito à comunicação entre configuradores em ambientes distribuídos, e à transparência de localização dos componentes envolvidos.

Foram realizados também testes com a transmissão de um fluxo de vídeo, utilizando a tecnologia JMF para o tratamento das questões relacionadas ao fluxo, e envolvendo a adaptação deste fluxo através da troca de qualidade do fluxo transmitido.

6.1. Trabalhos futuros

O modelo de interconexão apresentado foi proposto como um mecanismo de comunicação para fluxos de informações no contexto do *framework* Cosmos. Sua definição levou em consideração e influenciou os outros modelos definidos pelo *framework*, como o modelo de meta-componentes, e o modelo de gerenciamento de QoS.

A avaliação do modelo de interconexão foi realizada, em conjunto com os conceitos do *framework* Cosmos, no contexto de um protótipo com o objetivo de integrar os modelos desenvolvidos. Com isso, e considerando a abrangência do *framework*, várias questões não puderam ser tratadas no contexto dos modelos definidos, incluindo o modelo de interconexão apresentado neste trabalho.

Uma dessas questões diz respeito à detecção de fim de fluxo de dados de um canal de comunicação por parte da porta de entrada associada. Outra questão diz respeito à realização de adaptação dinâmica envolvendo a troca de componentes, que não foi abordada até o presente momento.

Com isso, o modelo de interconexão apresentado, em conjunto com os outros modelos definidos pelo *framework* Cosmos, abre diversas possibilidades de trabalhos futuros. Neste sentido, são enumeradas a seguir algumas proposições que podem ser exploradas no contexto de projetos futuros.

Dentre as perspectivas vislumbradas envolvendo o modelo de interconexão podemos citar:

A utilização de outras tecnologias de comunicação

A tecnologia de comunicação utilizada é definida pelos canais de comunicação. Uma idéia a ser explorada é a implementação de outras tecnologias de comunicação na forma dos canais de comunicação e a realização de testes envolvendo a comunicação *multicast* integrando diferentes tecnologias de comunicação simultaneamente;

A definição de estratégias que sinalizem a finalização de um fluxo de dados de um canal de comunicação para suas portas de entrada associadas.

A troca dos canais de comunicação que acontecem durante o processo de adaptação leva em consideração o término do fluxo de dados do canal primário. Para o presente trabalho assumiu-se que, quando o buffer da porta de entrada, correspondente ao canal primário, estiver vazio durante o processo de adaptação, o fluxo de dados foi finalizado. Entretanto, o fato do buffer de recepção estar vazio não garante que não existam mais dados em trânsito.

Para isso faz-se necessário a definição de uma estratégia que garanta às portas de comunicação o fim do fluxo de dados de um canal. Esta estratégia é dependente da tecnologia de comunicação utilizada pelo canal de comunicação.

Suporte a adaptação dinâmica envolvendo a troca de componentes

As experiências realizadas na definição e implementação do modelo de interconexão levaram em consideração a integração dos outros modelos definidos pelo *framework* Cosmos, resultando em uma implementação bastante simples oferecendo suporte a adaptação dinâmica no contexto de ajuste de parâmetros.

Neste sentido, um trabalho que pode ser desenvolvido consiste na avaliação e definição de políticas de adaptação envolvendo a troca de componentes.

A avaliação de desempenho do modelo de interconexão

Outro trabalho futuro proposto é a avaliação de desempenho do modelo de interconexão e dos componentes associados à implementação do *framework*. Nesta avaliação, devem ser considerados os retardos associados, por exemplo, com o tempo de troca de formato, tempo de troca de canal, ou mesmo de componentes.

A aplicação do modelo de interconexão em outros *middlewares* existentes;

Existem diversas propostas de *middlewares* reflexivos para sistemas distribuídos. Um possível trabalho futuro é a aplicação do modelo de interconexão proposto a outros *middlewares* existentes, utilizando a tecnologia de distribuição destes middleware para tratar a transparência de localização entre os componentes envolvidos.

7. Referências

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

[2] A. B. Lopes, G. Elias, M. Magalhães. Cosmos: *Um Framework para Configuração e Gerenciamento de Sistemas Multimídia Distribuídos Abertos*. In: Simpósio Brasileiro de Sistemas Multimídia e Web (WebMedia 2003) - III Workshop de Teses e Dissertações em Multimídia, Hipermídia e Web, 2003. Salvador, BA.

[3] A. B. Lopes, F. Borelli, G. Elias, M. Magalhães. *A Component-based Configuration and Management Framework for Open, Distributed Multimedia Systems*. In: 18th International Conference on Advanced Information Networking and Application (AINA'04), Fukuoka, Japão, 2004. pp 465-470.

[4] A. B. Lopes. *Um Framework para Configuração e Gerenciamento de Recursos e Componentes em Sistemas Multimídia Distribuídos Abertos*. Tese de Doutorado. Faculdade de Engenharia Elétrica e de Computação. Unicamp. 2006.

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

[6] C. E. Silva, A. B. Lopes, G. Elias, G. Lemos, M. Magalhães. *A Component Interconnection Model for Interactive Digital Television*. In: The IEEE 20th International Conference on Advanced Information Networking and Applications, 2006, Vienna. IEEE 20th International Conference on Advanced Information Networking and Applications. New York : IEEE Computer, 2006. v. 1. p. 959-964.

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

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

[9] L. Bergmans, 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.

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

[11] A. Andersen, G. S. Blair, and F. Eliassen. *A reflective component-based middleware with quality of service management*. In Protocols for Multimedia Systems (PROMS), Cracow, Poland, October 2000.

[12] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, K. Saikoski, "*The Design and Implementation of Open ORB V2*", IEEE Distributed Systems Online, Volume 2, Number 6, 2001.

[13] F. M. Costa, G. S. Blair. *Integrating Meta-Information Management and Reflection in Middleware*. In: 2nd International Symposium on Distributed Objects and Applications. (DOA'00), 2000, Antuérpia, Bélgica. DOA'00: International Symposium on Distributed Objects and Applications. Los Alamitos, CA, USA : IEEE Computer Society Press, 2000. p. 133-143.

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

[15] F. Siddiqui. *Component Based Software Engineering - A look at reusable software components*. http://www.smb.uklinux.net/reusability/; acessado em 28/07/2006.

[16] 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.

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

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

[19] A. S. Tanenbaum, M. V. Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.

[20] R. Koster. *A Middleware Platform for Information Flows*. PhD Thesis. Universitat Kaiserslautern. Kaiserslautern, 2002.

[21] G. Blair, J. B. Stefani. *Open Distributed Processing and Multimedia*. Addison-Wesley, 1998.

[22] E. Bouix, M. Dalmau, P. Roose, F. Luthon. *A Multimedia Oriented Component Model*. In: 19th International Conference on Advanced Information Networking and Applications, 2005. AINA 2005. Volume: 1, pp: 3- 8. ISSN: 1550-445X. ISBN: 0-7695-2249-1. DOI: 10.1109/AINA.2005.38.

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

[24] ITU-T/ISO. *Reference Model for Open Distributed Processing, Parts 1, 2, 3*. ITU-T X.901-X.904 | ISO/IEC IS 10746-(1, 2, 3). 1995.

[25] 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. pp 1258-1264.

[26] 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.

[27] 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.

[28] A. Lopes, C. Silva, G. Elias, M. Magalhães. *Um Modelo de Metacomponentes para Suporte à Adaptação Dinâmica em um Middleware para Sistemas de Televisão Interativa*. In: XII Simpósio Brasileiro de Sistemas Multimídia e Web - WEBMEDIA2006, 2006, Natal. Proceedings of the 12th Brazilian symposium on Multimedia and the web. New York : ACM Press, 2006. v. 1. p. 193-202.

[29] J. A. Medeiros. *Modelo de Metadados e Parser para uma Linguagem de Metaprogramação para o AdapTV*. Relatório de Graduação. DIMAp, UFRN, 2005.

[30] F. Amaro, F. Lopes, C. Silva, D. Oliveira, A. Lopes, G. Elias and G. L. Souza. *Especificação e Gerenciamento de QoS em um Middleware para Sistemas de Televisão Digital Interativa*. In: XI WebMedia - Simpósio Brasileiros de Sistemas Multimídia e Web, 2005, Poços de Caldas. Anais do XI Simpósio Brasileiro de Sistemas Multimídia e Web, 2005. p. 77-86.

[31] 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. 20th International Conference on Advanced Information Networking and Applications - Volume 1 (AINA'06), 2006, pp. 959-964.

[32] A. Lopes, F. Borelli, G. Elias, and G. Lemos. *Projeto e Implementação de um Middleware para sistemas de Televisão Digital Interativa*. In WebMedia 2004. In Proc. of the WebMedia & LAWeb 2004 Joint Conference, 2004. V2. pp. 96-103.

[33] Sun Microsystems. *Java Language*. Em http://java.sun.com.

[34] D. Oliveira. *Um Modelo de Recursos Virtuais para o Arcabouço Cosmos*. Relatório de Graduação. DIMAp, UFRN, 2005.

[35] Sun Microsystems. *JMF, Java Media Framework*. Em http://java.sun.com/products/jmf/.

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

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

[38] The GNU General Public License (GPL). June 1991. Disponível em: http://www.gnu.org/licenses/gpl.html ; Acessado em 28/07/2006.

[39] GNU Lesser General Public License (LGPL). February 1999. Disponível em: http://www.gnu.org/copyleft/lesser.html ; Acessado em 28/07/2006.

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

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo