PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

**Marcio Ferreira Moreno**

# Um Middleware Declarativo para Sistemas de TV Digital Interativa

**Dissertação de Mestrado**

Dissertação apresentada como requisito parcial para obtenção do título de Mestre pelo Programa de Pós-Graduação em Informática da PUC-Rio.

Orientador: Prof. Luiz Fernando Gomes Soares
Co-orientador: Prof. Rogério Ferreira Rodrigues

Rio de Janeiro, abril de 2006.

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

**Marcio Ferreira Moreno**

# Um Middleware Declarativo para Sistemas de TV Digital Interativa

Dissertação apresentada como requisito parcial para obtenção do título de Mestre pelo Programa de Pós-Graduação em Informática da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

**Prof. Luiz Fernando Gomes Soares**
Orientador
Departamento de Informática - PUC-Rio

**Prof. Rogério Ferreira Rodrigues**
Co-orientador
Departamento de Informática - PUC-Rio

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

**Prof. Sérgio Colcher**
Departamento de Informática - PUC-Rio

**Prof. José Eugênio Leal**
Coordenador Setorial do Centro
Técnico Científico – PUC-Rio

Rio de Janeiro, 12 de abril de 2006.

**Marcio Ferreira Moreno**

Graduou-se em Informática pela UFJF (Universidade Federal de Juiz de Fora) em 2004. Foi bolsista CNPq na área de Projetos Colaborativos assistidos por Redes durante a graduação.

Ficha Catalográfica

Moreno, Marcio Ferreira

Um Middleware Declarativo para Sistemas de TV Digital Interativa / Marcio Ferreira Moreno; orientador: Luiz Fernando Gomes Soares – Rio de Janeiro: PUC, Departamento de Informática, 2006.

105 f. ; 29,7 cm

Incluí referências bibliográficas.

1. Informática – Teses. 2. Middleware Declarativo. 3. TV Digital Interativa. 4. Sincronismo de Mídias. 5. Formatador. 6. NCL. 7. MPEG-2. 8. DSM-CC. 9. DirectFB. 10. Video4Linux. I. Soares, Luiz Fernando Gomes. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Este trabalho é dedicado

*Aos meus pais, Jasmina e Wanderley, sábios lutadores que sempre mostraram, com exemplos, o caminho certo a ser trilhado. Eternos amores da minha vida.*

*Aos meus irmãos, Patrícia e Marcelo, meus ídolos. Sinônimos de amor, amizade, companheirismo, e inteligência.*

*À Suzana, um novo amor que está às margens da perfeição.*

## Agradecimentos

Agradeço, em primeiro lugar, ao meu orientador, Professor Luiz Fernando Gomes Soares, por todos os valiosos ensinamentos e por não medir esforços na sua orientação. Pela sua perfeição como pesquisador e orientador. Concentro nele todas as minhas ambições como cientista.

Agradeço também ao meu chefinho LF, por não existir problema que seu vitaminado cérebro não resolva. Sem querer incomodar, mas já incomodando: obrigado chefinho!

Em especial agradeço ao meu co-orientador, Professor Rogério Ferreira Rodrigues, por sua dedicação ao trabalho e pela orientação. Suas características (companheirismo, atenção, paciência e inteligência) foram fundamentais na concepção deste trabalho.

Agradeço ainda ao meu amigo Roger, pelo companheirismo na feijoada com caipirinha aos sábados e domingos de trabalho e pela preciosa e paciente ajuda na exaustiva depuração de código através de *couts*. Esse é mais F. que o bolinha.

Agradeço também ao bolinha pelas inúmeras noites viradas. Enquanto eu dormia e, recentemente, namorava, lá estava o bolinha para fazer meu trabalho.

Em especial, agradeço ao meu irmão, Marcelo, e minha cunhada, Lorenza, pelo apoio na moradia, carinho, incentivo e bons conselhos.

Agradeço também aos amigos do TeleMídia, pacientes receptores dos meus ruídos, em especial ao Júnior (o que mais sofreu), ao Carlão (pelas inúmeras madrugadas de trabalho), a Cavendish (por ceder o Carlão) e a Lambão (pela ajuda com o exibidor Lua). Agradeço ainda aos professores e funcionários da PUC-Rio, pela qualidade do ensino e qualidade que trazem à esta universidade.

Agradeço também aos membros da banca, pelos comentários, sugestões e revisões.

Finalmente, agradeço à CAPES, a PUC-Rio e ao Laboratório TeleMídia pela estrutura, infra-estrutura e todo tipo de apoio, inclusive financeiro, fundamentais à realização deste trabalho.

## Resumo

Moreno, Marcio Ferreira. **Um Middleware Declarativo para Sistemas de TV Digital Interativa.** Rio de Janeiro, 2006. 105 páginas. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A evolução das técnicas de codificação digital, aliada aos esquemas eficientes de modulação para transmissões digitais, tornou possível o advento da TV digital (TVD). Entretanto, obter baixo custo nos terminais de acesso é fator crucial para o sucesso da TVD aberta, principalmente nos países em desenvolvimento. Para que o baixo custo comprometa o mínimo possível dos recursos dos terminais de acesso, é interessante que eles estejam isentos de custos adicionais como, por exemplo, software, propriedade intelectual e royalties. Um dos principais pontos para tornar isso possível concentra-se na escolha do middleware (que faz uso de mecanismos definidos por protocolos de comunicação, sistema operacional e suas bibliotecas) para suporte às aplicações. A maioria dos middlewares declarativos existentes privilegiam a interatividade em detrimento da sincronização. Entretanto, na maioria das vezes as aplicações de TVD devem lidar com a sincronização de objetos de diferentes tipos de mídia, além dos objetos de vídeo e áudio que compõem o fluxo principal. Assim, o sincronismo de mídias deve ser o foco da linguagem declarativa a ser utilizada pelo middleware, tratando a interatividade como um caso particular do sincronismo. Este trabalho tem como objetivo propor um middleware declarativo para sistemas de TVD interativa com foco no sincronismo de mídias. Na implementação do middleware proposto, a arquitetura modular do Formatador HyperProp, que serviu como base dessa implementação, foi reestruturada em um perfil simples, direcionado à TVD, e reimplementada na Linguagem C++. Todos os exibidores de mídia desenvolvidos atendem aos requisitos dos terminais de acesso de baixo custo.

## Palavras-chave

Middleware Declarativo; TV Digital Interativa; Sincronismo de Mídias; Formatador; NCL; MPEG-2; DSM-CC; DirectFB; Video4Linux.

## Abstract

Moreno, Marcio Ferreira. **A Declarative Middleware to Interactive TV Systems.** Rio de Janeiro, 2006. 105 pages. Master Thesis - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The evolution of digital modulation techniques and the efficient schemes for digital transmissions have allowed the advent of the digital TV. Conceiving low cost receivers is one of the main challenges to broaden digital TV use, mainly in non-developed countries. As a consequence, low cost requirements should not burden receiver resources that should try to reduce their costs in items such as software copyright and royalties. In this scenario, the middleware (which use mechanisms defined by the communication protocols, the operational system and its libraries) conception plays an important role.

A great number of declarative middlewares focuses on user interaction instead of synchronization, in its broad sense. However, the majority of digital TV applications deals with different types of media-object synchronization, beyond the audio and video that compose the main stream. Thus, the declarative middleware language focus should be placed on synchronism, having user interaction as a special synchronization case. This is the goal of this work.

The middleware implementation presented in this work is based on the modular architecture of the HyperProp Formatter, whose architecture was reorganized in a simple profile for digital TV systems. The implementation was carried out using C++ language, and all media players were developed to run in low cost receivers.

## Key words

Declarative Middleware; Interactive TV; Digital TV; Media Synchronism; Formatter; NCL; MPEG-2; DSM-CC; DirectFB; Video4Linux.

# Índice

## Lista de figuras

## Lista de tabelas

# 1
# Introdução

A evolução das técnicas de codificação digital, aliada aos esquemas eficientes de modulação para transmissões digitais, tornou possível o advento da TV digital. Atualmente, os sistemas de TV digital podem ser definidos, de forma resumida, como um conjunto de especificações que determinam as técnicas de codificação digital para transmitir o conteúdo de áudio, vídeo e dados, das emissoras (ou provedores de conteúdo) aos terminais de acesso dos telespectadores, e um conjunto de facilidades que dão suporte ao desenvolvimento de aplicações interativas.

No contexto de TV digital, terminal de acesso é o dispositivo responsável por tratar corretamente o sinal digital recebido, decodificando e exibindo de forma consistente a programação de TV, as aplicações e outros serviços avançados. Um terminal de acesso de TV digital, para oferecer um custo acessível, possui forte limitação na quantidade de recursos computacionais e hardware especializado para a exibição de conteúdo televisivo. Um terminal de acesso típico geralmente possui um processador de baixo custo para executar as tarefas das aplicações e do sistema operacional (possivelmente Linux ou Windows CE (O'Driscoll, 2000)) e pouca memória para uso das aplicações e do sistema operacional. Obviamente, podem existir terminais mais sofisticados com capacidade de processamento mais alta para suportar aplicações mais complexas, disco rígido com capacidade de armazenamento para gravar os programas transmitidos, ou mesmo armazenar aplicações oferecidas pelos provedores de conteúdo. Entretanto, melhoras como essas aumentam sensivelmente o custo do terminal de acesso.

O hardware de um terminal de acesso pode variar de acordo com o sistema de TV digital utilizado e seus requisitos de desempenho. Como exemplo, um terminal de acesso do sistema de TV digital europeu terrestre deve possuir hardware para decodificar transmissões de acordo com o padrão de modulação europeu COFDM (*Coded Orthogonal Frequency Division Mulitiplexing*), enquanto que um terminal do sistema americano terrestre deve possuir hardware

para decodificar transmissões de acordo com o padrão de modulação americano VSB (*Vestigial Side Band*) (Schwalb, 2004). Considerando que a maioria dos padrões de TV adotam codificação MPEG, em geral, os terminais de acesso possuem uma decodificadora MPEG para recuperar os sinais de áudio e vídeo principais da programação sintonizada.

Ainda nos terminais de acesso, é comum existir uma camada de abstração, denominada middleware, responsável por esconder das aplicações a complexidade dos mecanismos definidos pelos padrões, protocolos de comunicação e até mesmo sistema operacional do equipamento. Simplificadamente, as implementações de middleware devem oferecer as bibliotecas necessárias às aplicações através de uma API (*Application Program Interface*) bem definida.

A seguir, são apresentadas a motivação para realização deste trabalho, os objetivos a serem atingidos, bem como a organização deste documento.

## 1.1. Motivação

O primeiro padrão aberto utilizado como middleware de TV digital interativa foi definido pelo grupo MHEG (*Multimedia and Hypermedia Experts Group*) e publicado em 1997 pela ISO (*International Standards Organization*). O padrão oferece um paradigma declarativo para o desenvolvimento de aplicações hipermídia.

As primeiras especificações do grupo MHEG, denominadas MHEG-1 (ISO, 1997a), definem uma forma declarativa para a representação de objetos hipermídia por meio da notação base ASN.1 (*Abstract Syntax Notation One*) (ISO, 2002). MHEG-1 define um formato de intercâmbio para possibilitar que uma mesma representação dos objetos hipermídia seja apresentada em máquinas diferentes e ainda permite a associação de códigos procedurais aos objetos hipermídia. Para garantir a portabilidade entre diferentes plataformas dessa parte procedural embutida nos objetos, o grupo MHEG publicou as especificações

MHEG-3[1] (ISO, 1997b), que definem uma máquina virtual MHEG e uma representação em *bytecode* MHEG (Morris & Chaigneau, 2005).

Atualmente, com o objetivo de deixar as aplicações presentes nos terminais de acesso independentes de plataforma, a maioria das implementações de middleware lança mão do mesmo recurso utilizado pelo grupo MHEG: máquinas virtuais como ambiente de execução para as aplicações. Ou seja, uma máquina virtual do middleware cria um ambiente de execução comum às suas aplicações. Nesse cenário, as aplicações, após desenvolvidas em código-fonte, são compiladas para *bytecode* de sua máquina virtual não sendo, portanto, necessário manter diferentes versões das aplicações para cada plataforma de terminal de acesso contemplada.

Entretanto, o padrão MHEG-1 não foi bem sucedido no mercado. Seu mau resultado foi atribuído à complexidade dos conceitos definidos em suas especificações, bem como ao mercado não estar ainda preparado para o grande número de recursos oferecidos pelo padrão. Para resolver esses problemas, o grupo MHEG criou um perfil simplificado do padrão MHEG-1, publicando em abril de 1997 as especificações MHEG-5[2, 3] (ISO, 1997c).

Paralelamente, a tendência tecnológica Java ganhava força, ameaçando o MHEG e fazendo com que o grupo de padronização especificasse uma API Java para o padrão em 1998: o MHEG-6 (ISO, 1998a). O padrão MHEG-6 permite que as aplicações MHEG-5 sejam associadas a aplicações Java por meio de uma API (iso.mheg5), que mapeia a hierarquia de classes MHEG-5 em classes Java. Assim, MHEG-6 permite às aplicações Java acessarem as variáveis do domínio MHEG e obter resultados após computação realizada nesse domínio. Além disso, as aplicações Java podem manipular os objetos MHEG-5 de forma a controlar as apresentações desses objetos. Além de oferecer uma API Java para o MHEG-5, o

---

[1] O padrão MHEG-2 especificaria o SGML como linguagem-base para a codificação das representações definidas pelo MHEG-1, porém foi cancelado por falta de recursos.

[2] O padrão MHEG-4 define o procedimento oficial ISO para registro de identificadores usados nas outras partes do MHEG. Portanto, devem ser registrados no MHEG-4 os novos objetos relacionados ao MHEG que surgirem, modificações de valores na numeração existente, tabelas com eventos de entrada, atributos de conteúdo explicitamente definidos, como tabela de cores ou fontes, entre outros.

MHEG-6 possui uma especificação de máquina virtual Java, para garantir sua portabilidade.

Apesar de nunca ter sido desenvolvido um middleware conforme o padrão MHEG-6, suas especificações serviram de base para um padrão de middleware para TV digital interativa, publicado em 1998 e desenvolvido pelo consórcio de empresas DAVIC (*Digital Audio Visual Council*) [DAVIC99]. Esse consórcio representou diversos setores da indústria audiovisual e foi iniciado em 1994, mas extinto após cinco anos de atividade, conforme previsto em seu estatuto. No padrão DAVIC, novas APIs Java foram adicionadas ao MHEG-6, fazendo com que a linguagem Java tivesse mais acesso aos recursos do terminal de acesso como, por exemplo, controle da apresentação de conteúdo audiovisual por meio do arcabouço JMF (*Java Media Framework*) (Sun, 1999), e gerenciamento dos recursos do terminal de acesso.

A linguagem Java destacou-se durante o desenvolvimento dos padrões de TV digital europeu DVB (*Digital Video Broadcasting*) (DVB, 2004). O fato de algumas empresas que faziam parte do consórcio DAVIC, e tiveram experiências com as APIs Java, estarem envolvidas no desenvolvimento dos padrões DVB, contribuiu substancialmente para que, em 1999, surgisse a primeira implementação em Java sobre um padrão aberto de middleware. Sua implementação seguiu as especificações do middleware europeu MHP (*Multimedia Home Platform*) (ETSI, 2003) e significou a mudança do paradigma de programação declarativo para o paradigma orientado a objetos (ou de middleware declarativo para procedural, como é a denominação comum no mundo de TV digital).

O desenvolvimento de um middleware puramente procedural não supriu, entretanto, a necessidade de um middleware declarativo. Por oferecerem um nível de abstração mais elevado, as linguagens declarativas não exigem tanto conhecimento de programação quanto as linguagens procedurais.

Os middlewares declarativos são mais adequados para aplicações cujo foco casa com o objetivo específico para o qual a linguagem declarativa foi

---

[3] O padrão MHEG-5 foi o primeiro padrão aberto utilizado como middleware de TV digital interativa e ainda é utilizado pela principal rede de TV digital do Reino Unido.

desenvolvida; ao contrário do uso de middlewares procedurais que são de propósitos mais gerais e ideais para aplicações que precisam de uma linguagem com maior poder de expressão. Aos poucos os padrões de TV digital foram reincorporando as funções do middleware declarativo ao middleware procedural Java. O foco da linguagem declarativa deixou de ter a rica expressividade da linguagem MHEG, passando a ter a simplicidade excessiva da linguagem HTML (ISO, 2000b).

Atualmente, a maioria das implementações de middleware (ATSC, 2005); (DVB, 2004) oferece simultaneamente ambientes distintos para as aplicações, ou seja, um ambiente para controle das aplicações procedurais e um ambiente para controle das aplicações declarativas. Entretanto, isso é possível apenas em terminais que possuam poder de processamento suficiente para executar, ao mesmo tempo, duas máquinas virtuais de características bem distintas como, por exemplo, uma máquina virtual Java e uma máquina virtual MHEG. Na realidade, a presença de cada máquina virtual acarreta em aumento de custo do terminal de acesso, considerando não só o maior poder de processamento necessário para executá-la como os pagamentos sobre royalties e propriedades intelectuais.

O baixo custo do terminal de acesso é fator crucial para o sucesso da TV digital aberta, principalmente nos países em desenvolvimento. Para que o baixo custo comprometa o mínimo possível dos recursos dos terminais de acesso, é interessante que eles estejam isentos de custos adicionais como, por exemplo, software, propriedade intelectual e royalties. Um dos principais pontos para tornar isso possível concentra-se na escolha do middleware, que faz uso de mecanismos definidos por protocolos de comunicação e pelo sistema operacional e suas bibliotecas, para suporte às aplicações. No entanto, tal middleware deverá ter um foco mais geral, de forma a atender a maioria das aplicações a serem desenvolvidas.

Os middlewares declarativos existentes como, por exemplo, DVB-HTML (ETSI, 2003) e DASE declarativo (ATSC, 2003), possuem foco na interatividade, um tipo de sincronismo de mídias que facilita o desenvolvimento das aplicações com interação do telespectador. Assim, para esses middlewares, qualquer outro tipo de sincronismo de mídias, que não seja a interatividade, deve ser descrito de forma procedural.

A necessidade de sincronismo no contexto da TV digital, sem a interação do usuário telespectador, está presente mesmo em sua aplicação mais primária: a exibição temporalmente sincronizada do fluxo de vídeo e áudio principal de um programa. As aplicações para TV digital, na maioria das vezes, devem lidar com a sincronização, espacial e temporal, de objetos de diferentes tipos de mídia, além dos objetos de vídeo e áudio que compõem o fluxo principal. Assim, o sincronismo de mídias deve ser o foco da linguagem declarativa, que deve tratar a interatividade como um caso particular do sincronismo.

Uma abordagem possível para contemplar linguagens que vislumbram o sincronismo de mídias, mas que não resolve o problema da limitação de recursos da plataforma, foi desenvolvida pelo *Telecommunications Software and Multimedia Laboratory* da *Helsinki University of Technology*. Esse laboratório realizou recentemente a adaptação de um formatador SMIL (*Synchronised Multimedia Integrated Language*) (Bulterman, 2004), uma linguagem declarativa com foco em sincronismo de mídias, para o contexto da TV digital (Lamadon et al, 2003). Além de questões de interface com o usuário, métodos de interatividade para acessar o conteúdo multimídia como, por exemplo, a utilização do canal de retorno, foram considerados. O formatador SMIL para TV digital foi desenvolvido com a tecnologia Java para executar sobre o middleware MHP, consistindo em um Xlet (ETSI, 2003) que necessita de uma máquina virtual DVB-Java (ETSI, 2003), ou DVB-J, para controlar seu ciclo de vida (i.e., carregar, iniciar, parar e terminar) (Lamadon et al, 2003). Essa mesma abordagem foi também seguida pelas primeiras implementações do formatador do sistema HyperProp, adaptado para o ambiente de TV Digital.

Desenvolvido no Laboratório TeleMídia, o Formatador HyperProp (Rodrigues, 2003) consiste de uma ferramenta capaz de interpretar documentos especificados em NCL (*Nested Context Language*) (Muchaluat-Saade, 2003), uma das principais linguagens declarativas existentes com foco no sincronismo de mídias, baseada no modelo conceitual NCM (*Nested Context Model*) (Soares et al, 2003). A partir da especificação do documento NCL recebida, o Formatador HyperProp constrói um plano de apresentação que irá conter as características de apresentação de cada objeto de mídia, a programação das tarefas a serem escalonadas e as informações dos relacionamentos de sincronização entre os objetos de mídia. A sincronização inter-mídia direciona a sincronização intra-

objetos, que é realizada com o auxílio de mecanismos de reserva de recursos e também de um mecanismo de pré-busca, escalonado através de um plano de pré-busca construído a partir do plano de apresentação. Baseado nos eventos gerados pelos exibidores, nos eventos gerados pelo usuário e no plano de apresentação, o escalonador de apresentação controla a execução sincronizada de todo o documento, realizando ajustes quando esses se fazem necessários.

No formatador HyperProp, existe uma independência entre o controlador da apresentação e as ferramentas de exibição (módulos responsáveis pelo controle da exibição de cada objeto em si). Para isso, um modelo de interface, genérico e adaptável, para integração de ferramentas de exibição, é implementado, permitindo tratar igualmente todos os objetos sendo exibidos, independente da aplicação exibidora que os controla. O *framework* de integração das ferramentas permite que novas ferramentas de exibição venham a ser incorporadas, e outras tantas existentes possam ser adaptadas, funcionando de maneira integrada entre si e com o formatador hipermídia (Rodrigues et al, 2001). A versão atual do Formatador HyperProp é implementada em Java e, naturalmente, utiliza JMF para exibir conteúdos de mídia contínua.

Conforme discutido, além das preocupações com royalties e propriedade intelectual, implementações Java têm seus problemas ligados ao desempenho e recursos de máquina demandados. Mesmo hoje em dia as máquinas virtuais Java impõem perdas de desempenho. Para resolver essas e outras questões relacionadas ao contexto de TV digital, a implementação de um middleware declarativo independente do uso de tecnologia Java fez-se necessária.

## 1.2. Objetivos

Este trabalho tem como objetivo principal propor um middleware declarativo para sistemas de TV digital interativa. Entre outros requisitos ligados ao custo, esse middleware declarativo deve estar restrito às tecnologias que não acrescentam dispêndio através de propriedades intelectuais, patentes ou royalties. O objetivo principal deste trabalho condiz com os requisitos do Sistema Brasileiro de TV Digital (SBTVD).

Para promover a implementação do middleware proposto, a arquitetura modular do Formatador HyperProp, que serviu como base da implementação, foi

estendida e reestruturada para um perfil simples, direcionado à TV digital, e reimplementada na linguagem C++. Todos os exibidores de mídia (ferramentas responsáveis por controlar a exibição de cada objeto que compõe um programa de TV digital) foram também desenvolvidos, de forma a suportar os requisitos dos terminais de acesso (i.e. bibliotecas de código aberto com baixos requisitos de processamento). Além disso, outras questões relacionadas ao contexto da TV digital foram consideradas:

- Ao contrário da interação entre o usuário e os aplicativos para PC, usualmente realizada através do uso de um mouse, o middleware declarativo deve oferecer às aplicações uma interface com o usuário que possibilite a interação através de um dispositivo de controle remoto simples;
- No desenvolvimento de software para PC, a portabilidade não é uma questão difícil de se resolver devido ao uso ilimitado de máquinas virtuais. No entanto, para garantir portabilidade, o middleware declarativo deverá ser desenvolvido conforme o padrão ISO para a linguagem C++ (ISO, 2003), dessa forma, qualquer plataforma provida de script gnu g++ deve ser capaz de compilar, no mínimo, o núcleo do middleware declarativo;
- Ao contrário de usuários de computadores, os telespectadores não pensam na possibilidade de uma aplicação fechar por falta de recursos. O middleware declarativo deve possuir um gerenciamento de recursos de forma a ter o mínimo de consumo possível;
- Para a composição do middleware declarativo, ao novo formatador NCL devem ser integradas funções para recepção de objetos de mídia provenientes de fluxos MPEG-2 Sistemas (ISO, 2000a);
- Ainda com relação ao fluxo MPEG-2 Sistemas, deve ser dado suporte ao transporte de dados segundo a proposta DSM-CC (*Digital Storage Media – Command and Control*) (ISO, 1998b), e suporte a eventos de sincronização, cujo formato de dados deve fazer parte da especificação da API do middleware;
- O middleware declarativo deve possibilitar que edições das aplicações NCL sejam realizadas simultaneamente a suas apresentações. As modificações das aplicações declarativas,

especificadas no provedor de conteúdo por um ambiente de autoria, devem ser coerentemente atualizadas nos terminais de acesso, preservando todos os relacionamentos, incluindo aqueles que definem a estruturação lógica de um documento hipermídia (Moreno et al, 2005c).

## 1.3. Organização do Documento

O restante deste documento encontra-se organizado da seguinte forma. O Capítulo 2 apresenta discussões sobre os principais middlewares declarativos de TV digital existentes atualmente, realizando ainda uma análise comparativa. No Capítulo 3 são apresentadas as tecnologias relacionadas a esta dissertação. Uma atenção especial é dada às especificações do padrão MPEG-2, por ser o padrão para o transporte de conteúdo audiovisual mais utilizado nos sistemas de TV digital, bem como ao modelo de apresentação de interfaces gráficas adotado na maioria dos sistemas de TV digital, por sua importância no âmbito das aplicações para TV digital. O Capítulo 4, por sua vez, discorre sobre a arquitetura do middleware declarativo proposto. Para isso, são detalhadas as funcionalidades de cada um dos módulos pertencentes a essa arquitetura, assim como a forma de comunicação entre eles. O Capítulo 5 apresenta a descrição da implementação da arquitetura proposta no Capítulo 4. Por fim, o Capítulo 6 encerra a dissertação, descrevendo as conclusões obtidas a partir de todo trabalho realizado e os possíveis trabalhos futuros.

# 2
# Trabalhos Relacionados

Atualmente, entre os principais middlewares declarativos destacam-se o europeu DVB-HTML (ETSI, 2003), o americano DASE (*DTV Application Software Environment)* declarativo (ATSC, 2003) e o japonês BML (*Broadcast Markup Language*) (ARIB, 2002). As seções a seguir detalham as particularidades de cada um deles e oferece uma análise comparativa.

## 2.1.
## DVB-HTML

O middleware declarativo DVB-HTML faz parte das especificações do padrão MHP (ETSI, 2003), que foi desenvolvido com o objetivo de especificar um middleware, em conformidade aos padrões DVB, para terminais de acesso de TV digital interativa.

### 2.1.1.
### Arquitetura

A arquitetura do middleware MHP, ilustrada na Figura 1 (b), é baseada no uso de uma máquina virtual Java. O padrão define uma API genérica com o objetivo de oferecer acesso aos recursos dessa máquina virtual, bem como aos recursos do terminal de acesso. Uma aplicação Java que faz uso dessa API genérica é chamada de aplicação DVB-J. Como alternativa às aplicações DVB-J, o padrão MHP define (de forma opcional) uma aplicação, comumente denominada *user agent* na terminologia W3C[4], capaz de interpretar documentos declarativos hipermídia que reúnem um conjunto de padrões desenvolvidos para *Web*.

No contexto dos padrões DVB, uma aplicação interpretada pelo *user agent* é chamada de aplicação DVB-HTML. As especificações sobre o *user agent* DVB-

HTML e suas aplicações definem o middleware declarativo europeu DVB-HTML. Por ser opcional, um *user agent* pode ser implementado como um *plug-in* do middleware MHP, como será discutido na Seção 2.1.7. Finalmente, para controlar o ciclo de vida das aplicações DVB-J, bem como das aplicações DVB-HTML, o padrão MHP especifica um gerenciador de aplicações (ETSI, 2003).

A base do *user agent* DVB-HTML é a linguagem XHTML (*eXtensible HyperText Markup Language*) (Pemberton, 2002), versão baseada em XML da linguagem HTML. Por ser baseada em XML, a linguagem XHTML não possui tolerância a documentos HTML mal formados, isso significa que as implementações de um interpretador XHTML podem ser muito mais simples que as implementações de um interpretador HTML.

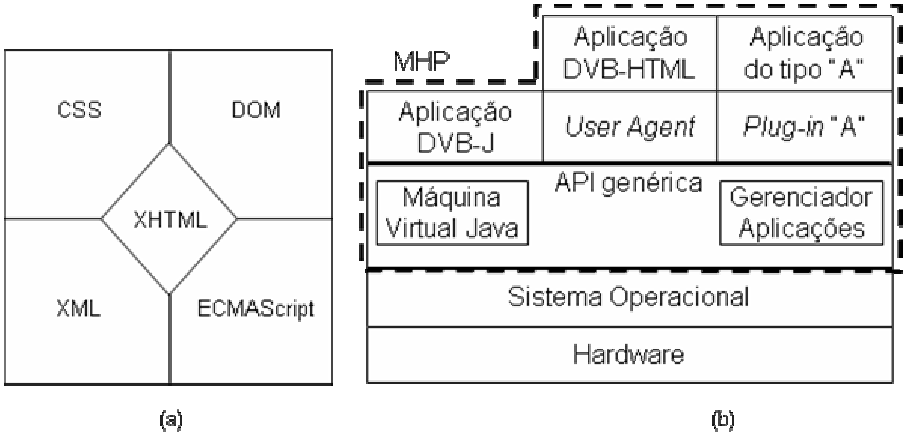


Figura 1: (a) Padrões reconhecidos pelo *User Agent* DVB-HTML.
(b) Arquitetura Middleware MHP (Procedural + Declarativo)

Além de XHTML, um *user agent* DVB-HTML deve dar suporte a outros padrões, conforme ilustra a Figura 1 (a), são eles: CSS (*Cascade Style Sheets*) (W3C, 1998), responsável pela formatação e layout de páginas HTML, ECMAScript (ECMA, 1999), responsável por adicionar recursos de programação (scripts) a documentos XHTML, e DOM (*Document Object Model*) (W3C, 2004), responsável por permitir que código procedural (por exemplo, os scripts) manipule estruturas e conteúdos de documentos XHTML. Esses padrões serão discutidos nas seções a seguir.

[4] http://www.w3c.org

### 2.1.2. Extensões sobre XHTML

A especificação W3C de XHTML define a linguagem através de módulos, apresentados na Tabela 1. Por ter foco em aplicações para *Web*, alguns desses módulos não possuem utilidade no contexto da TV digital. Assim, as especificações MHP definem para a linguagem DVB-HTML um subconjunto de XHTML, apresentado na Tabela 1, e estendem esse subconjunto através de especificações para um módulo adicional, denominado *DVB intrinsic events*. Os módulos XHTML que pertencem ao conjunto definido pelo padrão DVB são indicados na Tabela 1 através do símbolo "✓", por outro lado os módulos que não pertencem a esse conjunto são indicados com o símbolo "×". Outras tabelas apresentadas neste capítulo utilizam essa mesma notação.

| XHTML | DVB-HTML |
|---|---|
| Structure | ✓ |
| Text | × |
| Hypertext | ✓ |
| List | ✓ |
| Applet | × |
| Presentation | ✓ |
| Edit | × |
| Bidirectional text | ✓ |
| Basic forms | × |
| Forms | ✓ |
| Basic tables | × |
| Tables | ✓ |
| Image | ✓ |
| Client-side image map | ✓ |
| Server-side image map | × |
| Object | ✓ |
| Frames | ✓ |
| Target | ✓ |
| IFrame | ✓ |
| Intrinsic events | × |
| Meta-information | ✓ |
| Scripting | ✓ |
| Style sheet | ✓ |
| Style attribute | ✓ |
| Link | ✓ |
| Base | ✓ |
| Name identification | × |
| Legacy | × |

Tabela 1: Módulos XHTML utilizados por DVB-HTML.

O módulo *DVB intrinsic events* foi definido para gerar eventos de acordo com o estado da interpretação do documento DVB-HTML. Esse módulo oferece três tipos de eventos para os principais elementos de um documento DVB-HTML (*body* e *frame*): `ondvbdomstable`, `onload` e `onunload`. Esses eventos ocorrem de acordo com as precondições apresentadas na Tabela 2.

| Evento | Precondição |
|---|---|
| dvbdomstable | Quando a estrutura do documento DVB-HTML for completamente recebida e, depois, a interpretação (*parsing*) sobre esse mesmo documento tiver sido realizada (isso significa que a estrutura DOM do documento DVB-HTML está estável). Como conseqüência, uma modificação na estrutura DOM do documento DVB-HTML poderá ser realizada de forma "segura". |
| Load | Quando todos os recursos (imagens, Xlets, entre outros (ETSI, 2003)) relativos aos elementos do documento DVB-HTML forem recebidos . |
| unload | Quando o documento DVB-HTML for removido de uma janela ou *frame*. |

Tabela 2: Precondições dos Eventos Definidos no Módulo *DVB Intrinsic Events*.

### 2.1.3. Extensões sobre CSS

Com o objetivo de oferecer acesso à camada gráfica do modelo de apresentação do MHP, DVB-HTML requer conformidade com a recomendação W3C de CSS (W3C, 1998). Assim, o conjunto de regras e propriedades de CSS foi adaptado e estendido para atender às características de um ambiente de TV digital, sempre respeitando a conformidade com a recomendação W3C. As extensões foram especificadas com o objetivo de oferecer:

- **Integração com a camada gráfica**: uma regra, denominada `viewport`, foi criada com propriedades para permitir que uma aplicação DVB-HTML defina sua área de atuação na camada gráfica do modelo de apresentação. Uma das propriedades dessa regra é o `initial block`, que permite definir uma área retangular (com tamanho e posição) interna à `viewport`. Apenas uma regra `viewport` pode ser definida por aplicação. A regra `viewport`, bem como a propriedade bloco inicial são ilustradas na Figura Tabela 32;
- **Gerenciamento da opacidade**: a propriedade `opacity` foi criada para realizar efeitos de translucidez na camada gráfica do modelo de apresentação;
- **Integração com a camada de vídeo**: a propriedade `clip-video` foi criada para permitir que uma área retangular da camada de vídeo seja associada a elementos `object` ou `image` de um documento DVB-HTML. A Figura Tabela 32 apresenta um `clip-video` como uma área retangular equivalente à área do vídeo apresentado;

- **Navegação por controle remoto**: as propriedades `nav-up`, `nav-down`, `nav-left`, `nav-right`, `nav-index` e `nav-first` permitem alternar o foco entre elementos definidos no documento DVB-HTML através de eventos de controle remoto. Por exemplo, se ao especificar um elemento "X" no documento XHTML for atribuída, através de CSS, uma propriedade `nav-left` a "X", quando o evento `nav-left` (seta para esquerda, no padrão de tratamento de eventos de controle remoto (ETSI, 2003)) for gerado pelo controle remoto, o elemento "X" receberá o foco. Analogamente, outros elementos podem receber as propriedades `nav-up`, `nav-down`, e `nav-right`. A propriedade nav-first indica qual elemento deverá receber o foco quando a aplicação for inicialmente exibida e, finalmente, a propriedade `nav-index` atribui um botão específico (exceto aqueles definidos nas propriedades já citadas) do controle remoto ao elemento. Dessa forma, quando o botão atribuído for acionado, o respectivo elemento receberá o foco.

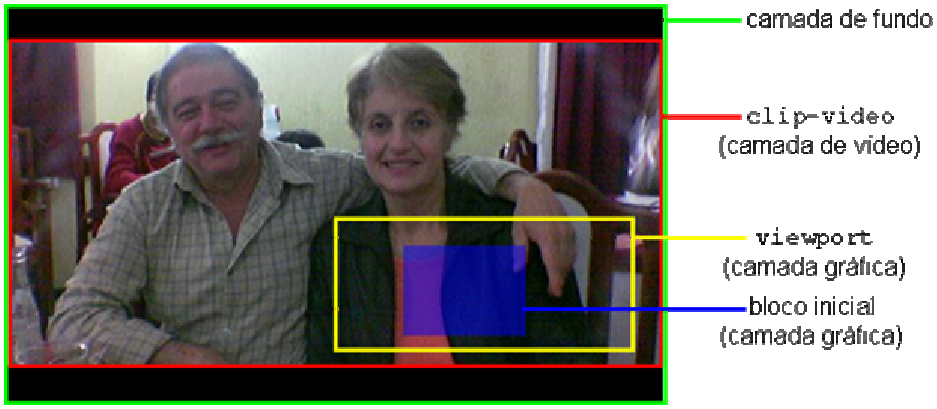


**Figura Tabela 32: Propriedades CSS sobre o Modelo de Apresentação MHP**

### 2.1.4. Extensões sobre DOM

Para permitir que o código procedural (na linguagem ECMAScript), presente em um documento DVB-HTML, ou mesmo outras aplicações (inclusive aplicações DVB-J, ou o próprio *user agent*), manipulem de forma dinâmica o

conteúdo de um documento DVB-HTML, o padrão MHP especifica o uso das recomendações W3C do modelo DOM.

Entre os principais objetivos do modelo DOM estão a criação de estruturas lógicas para documentos especificados em linguagens baseadas em XML, bem como a forma com que esses documentos serão acessados e manipulados por códigos procedurais como, por exemplo, Java e ECMAScript (W3C, 2004). Para isso, o modelo DOM especifica uma API (W3C, 2004).

O modelo DOM foi desenvolvido com foco em documentos hipermídia baseados em tecnologias desenvolvidas para Web. Assim, o padrão MHP especifica que o *user agent* DVB-HTML precisa oferecer suporte a apenas um subconjunto do modelo DOM, conforme descrito na Tabela 4.

| **Módulo DOM** | | **DVB-HTML** |
|---|---|---|
| **Pacote** | **String** | |
| Level 2 core | Core | ✓ |
| | XML | × |
| Level 2 HTML | HTML | × |
| Level 2 views | Views | ✓ |
| Level 2 style sheets | StyleSheets | × |
| Level 2 CSS style sheets | CSS | × |
| | CSS2 | ✓ |
| Level 2 events | Events | ✓ |
| | UIEvents | ✓ |
| | MutationEvents | ✓ |
| | HTMLEvents | × |
| | MouseEvents | × |
| Level 2 Traversal and Range | Traversal | × |
| | Range | × |

Tabela 4: Módulos DOM utilizados por DVB-HTML.

Além de utilizar os módulos recomendados pelo W3C, apontados pela Tabela 4, cinco novos módulos DOM são especificados no padrão MHP: *DVB-HTML*, *DVB Events*, *DVB Key Events*, *DVB CSS* e *DVB Environment*.

O módulo DOM *DVB-HTML* substitui o módulo DOM *HTML* e foi definido para refletir as modificações nas semânticas dos módulos XHTML tratados pelo *user agent* DVB-HTML, conforme discutido na Seção 2.1.2.

Com o objetivo de permitir que códigos procedurais (scripts no documento DVB-HTML e classes Java, incluindo as classes de aplicações DVB-J) controlem

o ciclo de vida das aplicações DVB-HTML, em conformidade com as especificações MHP, o módulo DOM *DVB Events* especifica a interface denominada *DVBLifecycleEvent*. Essa interface define eventos como, por exemplo, “iniciando aplicação”, “pausando aplicação”, “retomando aplicação”, entre outros (ETSI, 2003). O módulo DOM *DVB Events* possui ainda uma interface, denominada *trigger-event*, que contém atributos para descrever, no modelo DOM, um evento DSM-CC, uma ocorrência no tempo representada por uma estrutura de dados que será discutida na Seção 3.2.2. A interface *trigger-event* permite que o *user agent* notifique a ocorrência de um evento DSM-CC específico às aplicações DVB-HTML interessadas. Esse mecanismo de sincronismo será discutido na Seção 2.1.6.

O módulo DOM *DVB Key Events*, por sua vez, oferece uma interface para eventos de controle remoto, enquanto que o módulo DOM *DVB CSS* tem o objetivo de permitir que códigos procedurais acessem as regras e propriedades CSS estendidas pelo padrão MHP, discutidas na Seção 2.1.3. Finalmente, o módulo DOM *DVB Environment* possui o objetivo de permitir que códigos procedurais acessem variáveis de ambiente definidas em uma aplicação DVB-HTML.

O fato de uma aplicação DVB-J utilizar DOM para modificar o comportamento de uma aplicação DVB-HTML significa a existência de um meio de comunicação do middleware procedural para o middleware declarativo MHP. Esse meio de comunicação é um dos dois tipos definidos pelas especificações MHP. A comunicação no outro sentido, do middleware declarativo para o middleware procedural, será discutida na próxima seção. A utilização de um desses dois meios de comunicação é denominada *bridge*, no contexto MHP.

### 2.1.5. Código Procedural através de ECMAScript

As especificações MHP determinam que um *user agent* DVB-HTML deve prover suporte a ECMAScript, com o objetivo de incrementar o poder de expressividade das aplicações DVB-HTML. A linguagem ECMAScript teve como origem uma miscelânea de tecnologias, entre as mais conhecidas estão JavaScript (Nestscape) e JScript (Microsoft). Entretanto, ECMAScript possui mais recursos que suas precursoras, sendo uma linguagem de programação

interpretada, que utiliza o paradigma orientado a objetos e pode oferecer suporte a scripts em documentos XML (ECMA, 1999). ECMAScript possui uma propriedade, denominada `Packages`, que a permite acessar objetos Java de um pacote que ela conheça. Por exemplo, para um documento XML que possui ECMAScript acessar uma classe que faz parte do pacote `java.lang`, basta utilizar a seguinte construção ECMAScript: `Packages.java.lang` (Perror, 2001). O padrão MHP especifica que os pacotes e classes do middleware procedural devem ser conhecidos por ECMAScript. Assim, através de ECMAScript, as aplicações DVB-HTML podem utilizar os recursos do middleware procedural MHP, caracterizando um meio de comunicação do middleware declarativo para o middleware procedural. Analogamente, uma aplicação DVB-J pode se tornar "alcançável" ao ECMAScript de uma aplicação DVB-HTML.

### 2.1.6. Sincronismo

O protocolo DSM-CC, que será discutido na Seção 3.2, é o único mecanismo especificado no padrão MHP para realizar o sincronismo do comportamento das aplicações DVB-HTML com o conteúdo audiovisual transmitido pelo provedor de conteúdo. O mecanismo de sincronismo é utilizado de acordo com as discussões sobre eventos DSM-CC que serão realizadas no próximo capítulo.

Uma das principais funções do *user agent* DVB-HTML ao receber um documento, é realizar a interpretação do mesmo. Durante a interpretação, um modelo DOM desse documento é construído pelo *user agent*. Nesse processo, através de ECMAScript ou mesmo através de uma aplicação DVB-J a qual o documento DVB-HTML faça referência, uma chamada pode estar presente para cadastrar alguma função procedural (associada ao documento DVB-HTML) como *listener* de um determinado tipo de evento, conforme ilustrado pelo código da função setupEventListeners, na Figura 3 (ETSI, 2003).

```
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//DVB//DTD XHTML DVB-HTML 1.0//EN"
   "http://www.dvb.org/mhp/dtd/dvbhtml-1-0.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:dvbhtml="http://www.dvb.org/mhp">
   <head>
      <script type="text/ecmascript">
         // event listener declaration
         function handleEvent(evt) { /* Handle the event */ }

         // the listener is positioned on the document root node,
         // i.e. the html node
         function setupEventListeners () {
            var htmlNode = document.documentElement;
            htmlNode.addEventListener("myTriggerEvent", handleEvent, true);
         }
      </script>
   </head>
   <body dvbhtml:onload="setupEventListeners()">
   </body>
</html>
```

Figura 3: Cadastro de aplicação DVB-HTML em um evento DOM, que pode ser a tradução de um evento DSM-CC

A partir de uma especificação de eventos, chamada de *fábrica de eventos* (*event factory*), o middleware MHP pode ser capaz de mapear eventos DSM-CC em eventos DOM, e dessa forma ter as aplicações DVB-HTML sincronizadas com os fluxos elementares transmitidos pelas emissoras. No exemplo, um evento DSM-CC poderia ser associado ao evento DOM denominado "*myTriggerEvent*". Além disso, o documento DVB-HTML interpretado pode ter um elemento que possui uma função procedural (através de ECMAScript ou mesmo através de uma aplicação DVB-J a qual o documento DVB-HTML faça referência) que cria um evento da interface DOM *trigger-event*. Essa interface, além de possuir atributos para descrever um evento DSM-CC no modelo DOM, conta ainda com um atributo, denominado *target*, que contém um evento DOM a ser gerado. A estrutura DOM do documento interpretado passa a possuir então um elemento *trigger-event*, com as informações (atributos) necessárias para mapear um evento DSM-CC em um evento DOM. Ou seja, quando um evento DSM-CC ocorrer, o *user agent* notificará as aplicações DVB-HTML interessadas (aquelas registradas, através de código procedura, para o tipo de evento DSM-CC que está ocorrendo). A notificação consiste em acessar o elemento *trigger-event* que corresponde ao evento DSM-CC que está ocorrendo e gerar o evento DOM presente no atributo *target* desse elemento.

### 2.1.7. Alternativas de Uso

Um *user agent* pode ser implementado como um *plug-in* do middleware procedural, a ser enviado pelo provedor de conteúdo, como um Xlet, quando for necessária a apresentação de uma aplicação DVB-HTML. Um *plug-in* é um conjunto de funcionalidades que podem ser adicionadas a uma plataforma MHP a fim de possibilitar a interpretação de formatos de conteúdo de aplicações ainda não suportadas nessa plataforma. Uma vez que o *plug-in* está em estado operacional, a plataforma deverá comportar-se como se o formato do conteúdo já fosse suportado de modo nativo, sem o uso do *plug-in*.

É importante notar que as especificações MHP não possibilitam a implementação de um ambiente que possua apenas o middleware declarativo DVB-HTML. Ele é considerado, pelo padrão DVB, dependente do middleware procedural MHP. O *user agent* utiliza a API genérica definida pelo middleware procedural, ou seja, um *user agent* tem acesso aos recursos do terminal de acesso, ou mesmo da máquina virtual Java, através da API procedural MHP. Além disso, o gerenciador de aplicações, que também foi definido no middleware procedural, é responsável por controlar o ciclo de vida das aplicações DVB-HTML. Segundo (MHP, 2005), é impossível a implementação de um ambiente puramente declarativo DVB-HTML em conformidade com as especificações MHP. O objetivo é evitar uma fragmentação no mercado, garantindo que o MHP seja sempre oferecido, também nas versões futuras, de forma compatível (MHP, 2005).

## 2.2. DASE Declarativo

O padrão DASE foi desenvolvido nos Estados Unidos pelo grupo ATSC[5] (*Advanced Television Systems Comittee*), tendo sua primeira versão sido concluída em 2002.

[5] http://www.atsc.org

### 2.2.1. Arquitetura

A arquitetura DASE, ilustrada na Figura 4, possui uma camada superior formada por aplicações DASE, que podem ser declarativas ou procedurais, utilizando tecnologias como XHTML, CSS, Java, JMF, entre outras. Tais tecnologias são disponibilizadas pelo middleware DASE, que é dividido em quatro módulos.

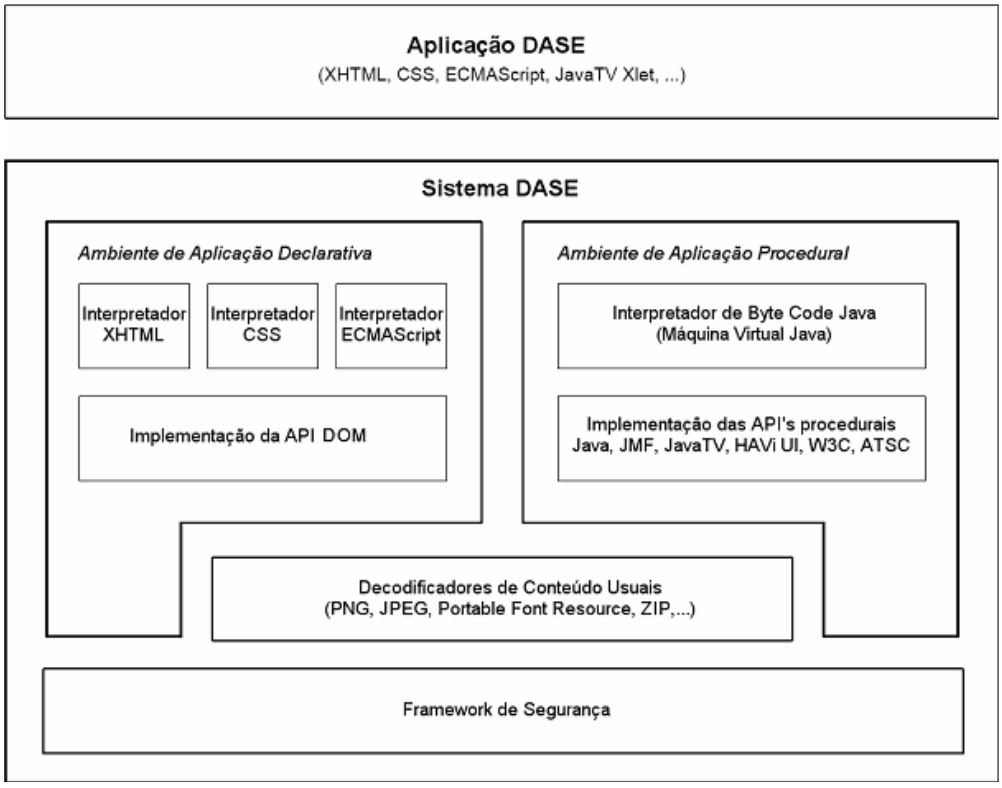


Figura 4: Arquitetura do Middleware DASE (Declarativo + Procedural)

O módulo do Ambiente de Aplicação Declarativa (DAE – *Declarative Application Environment*) consiste na parte declarativa do middleware DASE, sendo um subsistema lógico que, de forma semelhante ao middleware declarativo DVB-HTML, interpreta documentos XHTML com folhas de estilos CSS, scripts ECMAScript e possui suporte a DOM. Assim, uma aplicação que faz uso do DASE declarativo utiliza uma linguagem, oferecida por ele, para ter acesso aos recursos do terminal de acesso. Analogamente, o módulo do Ambiente de Aplicação Procedural (PAE – *Procedural Application Environment*) oferece sua API baseada na linguagem Java às aplicações procedurais. O módulo Decodificadores de Conteúdo atende às necessidades tanto do ambiente de

aplicação procedural quanto do ambiente de aplicação declarativa. Os seguintes decodificadores podem estar presentes nesse módulo: PNG, JPEG, ZIP, entre outros (ATSC, 2003). Finalmente, o módulo Framework de Segurança é responsável por questões de criptografia dos dados transmitidos entre o provedor do conteúdo e o terminal de acesso do usuário (ATSC, 2003).

### 2.2.2. Extensões sobre XHTML

As especificações DASE definem uma linguagem, denominada XDML (*Extensible DTV Markup Language*), que consiste em um subconjunto dos módulos XHTML recomendados pelo W3C. Os módulos selecionados por DASE são apontados na Tabela 5. Nenhuma extensão XHTML foi especificada pelo padrão DASE.

| XHTML | XDML |
|---|---|
| Structure | ✓ |
| Text | ✓ |
| Hypertext | ✓ |
| List | ✓ |
| Applet | × |
| Presentation | ✓ |
| Edit | × |
| Bidirectional text | ✓ |
| Basic forms | × |
| Forms | ✓ |
| Basic tables | × |
| Tables | ✓ |
| Image | × |
| Client-side image map | ✓ |
| Server-side image map | × |
| Object | ✓ |
| Frames | ✓ |
| Target | ✓ |
| IFrame | × |
| Intrinsic events | ✓ |
| Meta-information | ✓ |
| Scripting | ✓ |
| Style sheet | ✓ |
| Style attribute | ✓ |
| Link | × |
| Base | × |
| Name identification | ✓ |
| Legacy | × |

Tabela 5: Módulos XHTML utilizados por XDML.

### 2.2.3. Extensões sobre CSS

Com o objetivo de oferecer acesso à camada gráfica do modelo de apresentação DASE, suas especificações exigem conformidade com a recomendação W3C de CSS (W3C, 1998). Assim, de forma semelhante ao middleware declarativo europeu, um conjunto de regras e propriedades CSS foi definido para atender às características de um ambiente de TV digital, sempre

respeitando a conformidade com a recomendação W3C. As extensões definidas no padrão DASE, entretanto, são mais restritas que as especificadas pelo MHP. Apenas duas regras e uma propriedade foram definidas como extensão: uma regra para navegação por controle remoto (especificada da mesma forma que em MHP, discutida na Seção 2.1.3); uma propriedade para atualização dinâmica dos elementos de um documento XDML, denominada `atsc-dynamic-refresh` – a propriedade `atsc-dynamic-refresh` permite especificar quais elementos de um documento XDML (i.e. uma imagem, um objeto, ou mesmo todo o documento) devem ser dinamicamente atualizados quando um recurso qualquer da aplicação declarativa (i.e. qualquer elemento da aplicação descrito por XDML, CSS ou ECMAScript) em execução for atualizado; e, finalmente, uma regra para especificação de cores e opacidade de elementos, denominada `atsc-rgba` (vermelho, verde, azul e a porcentagem de opacidade).

### 2.2.4. Extensões sobre DOM

DOM foi adotado em DASE com os mesmos objetivos do padrão MHP: permitir que scripts (na linguagem ECMAScript), ou código procedural de outras aplicações (aplicações procedurais DASE, bem como ECMAScript de outras aplicações declarativas), manipulem de forma dinâmica o conteúdo de um documento declarativo DASE, caracterizando-se em um meio de comunicação do middleware procedural DASE para o middleware declarativo. Um subconjunto do modelo DOM foi especificado pelo padrão DASE, e modificado com a definição de novas interfaces específicas para TV digital. A Tabela 6 apresenta quais módulos do modelo DOM têm suporte no DASE declarativo.

As principais modificações feitas pelas especificações DASE concentram-se em definir interfaces adicionais em dois módulos DOM: *Core* e *View*. No módulo *Core*, a interface *DOMExceptionExt* acrescenta dois tipos de exceções, VALIDATION_ERR e NO_CLOSE_ALLOWED_ERR. A exceção VALIDATION_ERR ocorre quando uma aplicação tenta modificar um documento de forma a torná-lo inválido. Já a exceção NO_CLOSE_ALLOWED_ERR ocorre quando uma aplicação tenta fechar uma janela que não foi criada por ela. No módulo *View*, a interface *DocumentViewExt* foi criada para definir o espaço de coordenadas para as aplicações DASE

declarativas, ou seja, determinar características (resolução, posição, entre outras (ATSC, 2003)) da camada gráfica do modelo de apresentação DASE.

| Módulo DOM | | DASE declarativo |
|---|---|---|
| **Pacote** | **String** | |
| Level 2 core | Core | ✓ |
| | XML | ✓ |
| Level 2 HTML | HTML | ✓ |
| Level 2 views | Views | ✓ |
| Level 2 style sheets | StyleSheets | ✓ |
| Level 2 CSS style sheets | CSS | ✓ |
| | CSS2 | × |
| Level 2 events | Events | ✓ |
| | UIEvents | ✓ |
| | MutationEvents | ✓ |
| | HTMLEvents | ✓ |
| | MouseEvents | ✓ |
| Level 2 Traversal and Range | Traversal | × |
| | Range | × |

Tabela 6: Módulos DOM utilizados por DASE.

### 2.2.5. Código Procedural Através de ECMAScript

As especificações DASE definem ECMAScript com o mesmo objetivo do middleware declarativo DVB-HTML: incrementar a expressividade na construção das aplicações declarativas, permitindo que essas aplicações utilizem recursos de uma linguagem procedural, bem como do middleware procedural DASE, caracterizando um meio de comunicação do middleware declarativo para o middleware procedural. Da mesma forma que em DVB-HTML, uma aplicação DASE procedural pode se tornar "alcançável" ao ECMAScript de uma aplicação DASE declarativa, caracterizando a existência de aplicações híbridas.

### 2.2.6. Sincronismo

O mecanismo para realizar o sincronismo do comportamento das aplicações declarativas DASE com o conteúdo audiovisual transmitido pelo provedor de

conteúdo é baseado nos eventos DSM-CC. Entretanto, algumas modificações foram realizadas através das especificações A/93 (ATSC, 2002).

O padrão A/93 define duas entidades: *triggers* e *targets*. Os *triggers* são estruturas de dados, baseadas no descritor de eventos DSM-CC, utilizadas para especificar o momento de disparo de um alvo (uma aplicação DASE, por exemplo). Esse alvo, denominado *target*, é identificado dentro do *trigger* por meio de uma referência. Os *triggers* e *targets* podem ser enviados pelo provedor de conteúdo através de um carrossel de objetos (esse mecanismo será discutido na Seção 3.2.1).

Um *trigger* é formado basicamente por quatro campos: um identificador do *trigger*; uma referência temporal para o instante de "disparo" do *trigger* (que pode ser instantânea, como nos eventos "*do it now*", que serão discutidos na Seção 3.2.2); uma referência para o *target*; e dados específicos para o *target*. Assim, no mecanismo de sincronismo DASE, não é necessário que uma aplicação se registre como *listener* a um determinado tipo de evento (como em DVB-HTML).

Além de aplicações DASE, o campo *target* pode referenciar um evento DOM definido na estrutura lógica do documento DASE. De forma semelhante a DVB-HTML, ao interpretar um documento declarativo DASE, o DAE gera uma estrutura DOM para o mesmo. Note que o mecanismo definido nas especificações A/93 (ATSC, 2002) simplifica a forma de mapear um evento DSM-CC em um evento DOM, comparando com o mecanismo definido em DVB-HTML. Isso porque, ao contrário das aplicações DVB-HTML, as aplicações declarativas DASE não se preocupam com os tipos de eventos DSM-CC que estão sendo gerados, uma vez que elas são alvos (*targets*) desses eventos.

### 2.2.7. Alternativas de Uso

Diferentemente do middleware declarativo europeu, o DASE declarativo pode ser utilizado em plataformas que não possuem o DASE procedural instalado (CENELEC, 2003; Whitaker & Benson, 2003). As especificações DASE não definem o conceito de *plug-ins*. Entretanto, baseado na definição de uma classe Xlet, deve ser possível enviar uma implementação Xlet do DASE declarativo, com o objetivo de interpretar aplicações DASE declarativas em terminais que, originalmente, não possuem o ambiente declarativo (DAE) implantado.

Finalmente, o middleware declarativo DASE pode ser utilizado também de forma conjunta com o middleware procedural.

## 2.3. BML

O padrão japonês de TV digital, ISDB (*Integrated Services Digital Broadcasting*), foi desenvolvido em 1999 pela ARIB (*Association of Radio Industries and Broadcast*), um grupo de empresas, fabricantes e operadoras de televisão e de telecomunicações, fomentado pelo governo japonês. Seu desenvolvimento deu-se como segundo movimento de um trabalho iniciado em 1995 com o objetivo de digitalizar todos os sistemas de transmissão de televisão. O middleware japonês, também chamado de ARIB, foi definido através das especificações desse padrão. Diferentemente dos middlewares MHP e DASE, o middleware japonês foi inicialmente concebido contendo apenas um ambiente declarativo. Posteriormente, foi definido no padrão ARIB um middleware procedural (ARIB, 2002).

### 2.3.1. Arquitetura

De forma semelhante à arquitetura do middleware DASE, a arquitetura do middleware japonês, ilustrada na Figura 5, é composta por um ambiente procedural e um ambiente declarativo, independentes. O middleware procedural é baseado no uso de uma máquina virtual Java; o padrão que especifica o middleware procedural (ARIB, 2004) define uma biblioteca genérica com o objetivo de oferecer acesso aos recursos dos terminais. De forma semelhante ao middleware procedural europeu, *plug-ins* podem ser enviados pelo provedor de conteúdo a fim de possibilitar a interpretação de formatos de conteúdo de aplicações ainda não suportadas pela plataforma. O padrão do middleware declarativo (ARIB, 2002) define uma linguagem de marcação, denominada BML (*Broadcast Markup Language*), que, assim como as linguagens DVB-HTML e XDML, é baseada em XHTML, com suporte a CSS2, ECMAScript e DOM. Algumas extensões, bem como algumas restrições, foram especificadas pelo padrão. Essas especificações serão discutidas na próxima seção.

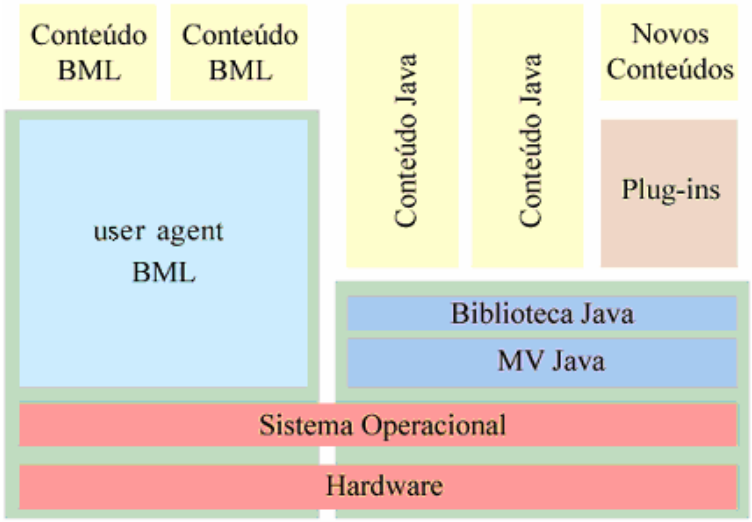


Figura 5: Arquitetura do Middleware ARIB (Declarativo + Procedural)

### 2.3.2. Extensões sobre XHTML

A linguagem BML consiste em um subconjunto dos módulos XHTML recomendados pelo W3C. Os módulos selecionados no padrão japonês são destacados na Tabela 7. Um único módulo de extensão, denominado *módulo BML*, foi especificado pelo padrão. Os principais elementos definidos nesse módulo são discutidos a seguir.

| XHTML | XDML |
|---|---|
| Structure | ✓ |
| Text | ✓ |
| Hypertext | ✓ |
| List | ✓ |
| Applet | × |
| Presentation | ✓ |
| Edit | ✓ |
| Bidirectional text | ✓ |
| Basic forms | ✓ |
| Forms | ✓ |
| Basic tables | ✓ |
| Tables | ✓ |
| Image | ✓ |
| Client-side image map | ✓ |
| Server-side image map | ✓ |
| Object | ✓ |
| Frames | ✓ |
| Target | ✓ |
| IFrame | ✓ |
| Intrinsic events | ✓ |
| Meta-information | ✓ |
| Scripting | ✓ |
| Style sheet | ✓ |
| Style attribute | ✓ |
| Link | ✓ |
| Base | ✓ |
| Name identification | × |
| Legacy | × |

Tabela 7: Módulos XHTML utilizados por BML.

Para permitir que o sincronismo entre o comportamento de uma aplicação BML e programas transmitidos pelo provedor de conteúdo sejam especificados através de código declarativo, BML define dois elementos para o controle de eventos: `bevent` e `beitem`. O elemento `bevent` possui todos os elementos `beitem` que serão processados no documento. Um elemento `beitem`, por sua vez, possui a capacidade de associar, através de seus atributos, código procedural à ocorrência de um evento específico. Cada elemento `beitem` é identificado através de um atributo chamado de *id*. Além disso, um elemento `beitem` possui um atributo *type*

que especifica o tipo de evento ao qual o elemento se refere. Os principais tipos de eventos, e suas respectivas semânticas, são apresentados na Tabela 8. Um terceiro atributo do elemento `beitem`, denominado *onoccur*, possui a identificação da chamada ao código procedural que será executado quando o evento específico ocorrer.

| Tipo de Evento | Semântica |
|---|---|
| EventMessageFired | Notifica a ocorrência de um determinado evento DSM-CC transmitido pelo provedor de conteúdo. |
| EventFinished | Notifica a ocorrência do fim de um programa de TV. |
| EventEndNotice | Pré-notifica a ocorrência do fim de um programa de TV. |
| Abort | Notifica que a apresentação de um conteúdo foi abortada. |
| ModuleUpdated | Notifica que foi detectada uma versão atualizada de um módulo específico de um determinado carrossel. |
| CCStatusChanged | Notifica que a linguagem da legenda sendo exibida foi alterada. |
| MainAudioStreamChanged | Notifica que um novo fluxo de áudio principal foi selecionado. |
| NPTReferred | Notifica que existe a possibilidade de adquirir a referência temporal corrente. |
| MediaStopped | Notifica que a decodificação de uma mídia específica (vídeo, áudio, caracteres, imagem estática, entre outras (ARIB, 2002)) foi completamente realizada. |
| MediaStarted | Notifica que a decodificação de uma mídia específica foi iniciada. |
| MediaRepeated | Notifica que a decodificação de uma mídia específica foi reiniciada. |
| DataButtonPressed | Notifica que um botão específico foi pressionado. |

Tabela 8: Tipos de eventos `beitem` e suas respectivas semânticas.

Além dos elementos para controle de eventos, o padrão ARIB define atributos para o elemento `object`, específicos para o controle da exibição dos fluxos transmitidos:

- ***streamstatus***: determina o estado de um fluxo. Os possíveis valores para esse atributo são "*stop*", "*play*" e "*pause*". O valor "*stop*" para um fluxo de áudio, por exemplo, significa na verdade um estado "*mute*";
- ***streamposition***: indica a posição temporal durante a exibição de um fluxo. Quando o fluxo está no estado "*pause*", o valor desse atributo é constante. Quando o fluxo está no estado "*stop*", é atribuído o valor zero para esse atributo. Finalmente, quando o fluxo está no

estado "*play*", o valor representa a unidade de tempo corrente do conteúdo de mídia em exibição (ARIB, 2002);

- ***streamlooping***: determina o número de vezes que um fluxo será exibido novamente;
- ***streamspeednumerator*** e ***streamspeeddenominator***: definem um fator, através de numerador e denominador, respectivamente, para a velocidade de exibição de um fluxo;
- ***streamlevel***: determina o volume de um fluxo. Note que esse atributo é principalmente útil para fluxos de áudio;
- ***streamstartposition*** e ***streamendposition***: determinam o início e o fim, respectivamente, de um fluxo que está armazenado no terminal de acesso.

A Figura 6 apresenta um exemplo que exibe um arquivo de áudio com o nome "sample.aiff", que deverá ser exibido dez vezes, a uma velocidade de exibição com fator "1/2", no volume máximo suportado pelo terminal de acesso. Note que o estado inicial do fluxo é "*stop*", isso porque um fluxo de áudio deve ser iniciado através de scripts.

```
<body>
  <object id="a" data = "sample.aiff"
   streamstatus = "stop"
   streamposition="0"
   streamspeednumerator = "1"
   streamspeeddenominator = "2"
   streamlooping = "10"
   streamlevel="100" />
</body>
```

Figura 6: Exemplo do Uso de Atributos para Controle de Exibição de Fluxo

### 2.3.3. Extensões sobre CSS

Com o objetivo de oferecer acesso ao modelo de apresentação ARIB, suas especificações exigem conformidade com a recomendação W3C de CSS (W3C, 1998), de forma semelhante aos middlewares declarativos MHP e DASE.

Além de propriedades para definir resolução e cores de elementos, algumas propriedades para controlar eventos de controle remoto foram definidas. Entre elas estão propriedades para seleção de foco entre elementos (nav-index, nav-up,

nav-down, nav-left e nav-right) especificadas da mesma forma que em MHP e DASE. Entretanto, uma propriedade interessante é definida no padrão ARIB, denominada used-key-list, para determinar os tipos de teclas que serão capturadas pelo *user agent* BML. Por exemplo, quando essa propriedade possui o valor "*basic*", os eventos de teclas numéricas do controle remoto são interpretados como seleção de um canal e não são considerados pelo *user agent* BML. Por outro lado, quando essa propriedade possui o valor "*numeric-tuning*", e um elemento que espera entrada de eventos do controle remoto possui o foco, os eventos de teclas numéricas do controle remoto são interpretados como uma entrada numérica para esse elemento e não como uma seleção de canais.

### 2.3.4. Extensões sobre DOM

DOM foi especificado em ARIB com objetivos semelhantes aos dos padrões MHP e DASE: permitir que o código procedural manipule, de forma dinâmica, o conteúdo de um documento BML. Um subconjunto do modelo DOM foi especificado pelo padrão ARIB, e modificado com a definição de novas interfaces. A Tabela 9 apresenta quais módulos do modelo DOM devem ter suporte, dado pelo *user agent* BML. As novas interfaces foram definidas apenas para refletir no modelo DOM as extensões sobre XHTML e CSS especificadas. Por exemplo, uma interface, denominada *BMLEvent*, possui atributos (ARIB, 2002) correspondentes à definição de um elemento `beitem`, discutido na Seção 2.3.2.

| Módulo DOM | | BML |
|---|---|---|
| **Pacote** | **String** | |
| Level 2 core | Core | ✓ |
| | XML | × |
| Level 2 HTML | HTML | ✓ |
| Level 2 views | Views | × |
| Level 2 style sheets | StyleSheets | × |
| Level 2 CSS style sheets | CSS | × |
| | CSS2 | ✓ |
| Level 2 events | Events | ✓ |
| | UIEvents | ✓ |
| | MutationEvents | ✓ |
| | HTMLEvents | × |
| | MouseEvents | × |
| Level 2 Traversal and Range | Traversal | × |
| | Range | × |

Tabela 9: Módulos DOM utilizados por BML.

### 2.3.5. Código Procedural Através de ECMAScript

As especificações ARIB definem ECMAScript com o objetivo de incrementar o poder de expressividade na autoria das aplicações BML. No contexto ARIB, ECMAScript é utilizado apenas para alterar dinamicamente o conteúdo de um documento BML, através da API DOM (Hori & Dewa, 2006). Ou seja, diferentemente de DVB-HTML e DASE declarativo, BML não define o uso de ECMAScript para acessar aplicações procedurais ou mesmo classes do middleware procedural japonês.

### 2.3.6. Sincronismo

Para realizar o sincronismo do comportamento das aplicações BML com os fluxos dos programas transmitidos pelo provedor de conteúdo, o padrão ARIB oferece ao autor de documentos uma abstração através de declarações BML. Essas declarações são formadas pelas extensões sobre a linguagem XHTML discutidas na Seção 2.3.2.

Ao interpretar um documento BML, o *user agent* gera uma estrutura DOM para o mesmo. Quando um elemento `beitem` é encontrado no documento, a estrutura DOM do documento interpretado passa a possuir um elemento *BMLEvent*, com as informações (atributos) necessárias para mapear o evento ocorrido a uma ação específica (atributo *onoccur*, discutido na Seção 2.3.2).

O *user agent* BML é responsável por monitorar a ocorrência dos tipos de eventos definidos na Tabela 8 (Seção 2.3.2), bem como notificar a ocorrência desses eventos às aplicações BML que possuam ações determinadas para esses eventos. Note que o mecanismo definido pelas especificações ARIB simplifica a forma de realizar o sincronismo do comportamento das aplicações com o conteúdo audiovisual transmitido, comparando com os mecanismos definidos em DASE declarativo e DVB-HTML, por exemplo, no caso da Figura 3, a função *setupEventListeners* seria substituída pelo código declarativo.

### 2.3.7. Alternativas de Uso

O middleware declarativo japonês é completamente independente de um middleware procedural, mas pode trabalhar em conjunto com o mesmo. Curiosamente, apesar de utilizar o mesmo conceito de *plug-in* do padrão europeu, o padrão ARIB não apresenta nenhuma especificação ou restrição para que seu middleware declarativo seja enviado como um *plug-in* do middleware procedural. Vale notar que o padrão japonês não define formas de comunicação entre o middleware declarativo e o procedural.

## 2.4. Análise Comparativa

Os principais middlewares declarativos existentes utilizam tecnologias desenvolvidas para *Web*, mais especificamente XHTML com suporte a CSS, DOM e ECMAScript. É um conjunto de tecnologias interessantes, mas ao mesmo tempo, segundo as discussões que serão realizadas a seguir, denotam os principais pontos fracos desses middlewares.

A linguagem XHTML favorece, geralmente, a autoria em função da simplicidade e da disseminação do HTML como linguagem para especificação de documentos na *Web*. Entretanto, as funcionalidades de XHTML são restritas

quando existe a necessidade de especificar o sincronismo e a interatividade, uma vez que XHTML restringe-se à formatação para apresentação e a definição de relações de referência entre documentos. Para superar essa limitação, as linguagens DVB-HTML e XDML adotam XHTML em conjunto com a linguagem ECMAScript. A linguagem BML, por sua vez, especifica extensões sobre XHTML, definindo marcações para descrever algum tipo de sincronismo. Entretanto, a ação dessas marcações geralmente são especificadas através de ECMAScript, exigindo que o autor programe, em detalhes, os passos que a aplicação deve executar para atingir seu objetivo. Além disso, outros relacionamentos de sincronização espaço-temporal, que não se encaixam nas extensões definidas por BML, também devem ser definidos através do uso de ECMAScript, por meio de uma especificação embutida, direta ou indiretamente, em um objeto XHTML. Como toda linguagem procedural, scripts apresentam um propósito mais geral, oferecendo uma maior flexibilidade para construção dos programas. No entanto, o nível de abstração oferecido para os autores é mais baixo, obrigando que os criadores de documentos hipermídia, como os programas de TV interativa, trabalhem, na realidade, como programadores de software.

Dada sua natureza de propósito geral, scripts não apresentam mecanismos diretos para a especificação de sincronismo, exigindo mecanismos de mais baixo nível para a sincronização entre objetos. Isso torna a programação em scripts mais flexível, porém suscetível a falhas quando há mudanças nas estruturas de dados em diferentes partes do escopo das funções. É também difícil identificar pelo código as várias estruturas de sincronismos utilizadas pelo autor. Tudo isso torna extremamente difícil modularizar e manter uma apresentação usando esse paradigma. Quando se deseja reusar parte de uma apresentação, não é claro onde uma cópia deve começar ou terminar. Especificação por script é geralmente útil para apresentações pequenas, mas a manipulação e edição de grandes documentos torna-se difícil sem a definição de uma estruturação. Além disso, a especificação detalhada de atividades paralelas apresenta os mesmos problemas da maioria das linguagens de programação.

Nas linguagens DVB-HTML, XDML e BML, as características de apresentação espacial dos objetos de mídia podem ser especificadas de forma independente através do uso de CSS. Entretanto, baseado em experiências com os navegadores existentes, é complexo conseguir o mesmo resultado na apresentação

de um mesmo documento com CSS em implementações diferentes (Morris & Chaigneau, 2005). Por exemplo, um mesmo documento com CSS interpretado pelos navegadores Mozilla e Internet Explorer, possivelmente, apresentará resultados distintos (apresentação de seus elementos dispostos de maneira diferente). Além disso, segundo (Morris & Chaigneau, 2005), aplicações que utilizam CSS demandam consumo de recursos elevado, levando a apresentações lentas em plataformas onde esses recursos são escassos.

No contexto geral, as linguagens DVB-HTML, XDML e BML utilizam principalmente o protocolo DSM-CC para sincronizar o comportamento de suas aplicações com os fluxos de programas transmitidos pelas emissoras. A sincronização é realizada de acordo com uma semântica embutida em eventos DSM-CC e um mapeamento desses eventos para eventos das linguagens declarativas. Esse é um mecanismo relativamente complexo, quando comparado a linguagens declarativas que vislumbram o sincronismo de mídias.

O fato das limitações dos principais middleware declarativos serem atribuídas às linguagens definidas por eles demonstra a importância de uma escolha criteriosa da linguagem declarativa. A linguagem NCL (Muchaluat-Saade, 2003) é uma alternativa interessante, por ser uma linguagem que vislumbra o sincronismo de mídias, baseada em um modelo conceitual que soluciona algumas limitações do modelo conceitual das linguagens baseadas em HTML (Soares, 2004). Através de um paradigma de causalidade/restrição (Soares et al, 2003), NCL possibilita a especificação das relações de sincronização temporal e espacial entre os objetos dispensando o uso de scripts e ao mesmo tempo oferecendo uma rica expressividade.

# 3
# Tecnologias Relacionadas

No contexto da TV digital, os equipamentos das emissoras e os terminais de acesso devem compartilhar de protocolos bem definidos para que o intercâmbio de informações e a provisão de serviços sejam possíveis. As seções a seguir detalham padrões e protocolos utilizados com esse propósito, o modelo de apresentação comumente adotado em sistemas de TV digital, bem como uma discussão sobre sistemas operacionais para terminais de acesso. Uma atenção é dada à biblioteca DirectFB pela sua importância neste trabalho.

## 3.1.
## MPEG-2 Sistemas

O padrão MPEG-2 *Systems* (ISO, 2000a), ou MPEG-2 Sistemas, estabelece como um ou mais sinais de áudio e vídeo, assim como outros dados (imagens estáticas, texto etc.), devem ser combinados de forma a serem transmitidos ou armazenados apropriadamente.

O MPEG-2 Sistemas define um fluxo elementar (ES – *Elementary Stream*) como um fluxo gerado pela codificação do conteúdo de vídeo, áudio ou dados específicos. Por ser um fluxo contínuo de informação, um ES pode apresentar dificuldades em sua manipulação (i.e. transmissão, armazenamento, entre outras (ISO, 2000a)). Assim, um ES costuma ser dividido em pacotes, gerando um novo fluxo chamado de PES (*Packetized Elementar Stream*).

As especificações MPEG-2 determinam o termo programa, chamado de serviço no contexto da TV digital, como um grupo composto de um ou mais fluxos PESs, correspondendo a sinais de vídeo, áudio e dados. Lembrando que, atualmente, a maioria dos discos DVDs utilizam codificação MPEG-2 Sistemas, o conteúdo desses discos é um bom exemplo de programa a ser citado.

A partir da definição de serviço, define-se dois formatos de multiplexação de fluxos no MPEG-2 Sistemas: o Fluxo de Transporte e o Fluxo de Programa. O Fluxo de Transporte pode conter vários serviços simultaneamente. Cada serviço

pode ter uma base de tempo diferente. O Fluxo de Programa só pode conter um serviço. Independente do formato de multiplexação, todos os fluxos elementares pertencentes a um mesmo serviço utilizam a mesma base de tempo. O formato de multiplexação utilizado pelos principais sistemas de TV digital existentes é o Fluxo de Transporte.

Simplificadamente, multiplexar serviços em um Fluxo de Transporte significa organizar os pacotes dos vários fluxos PESs, pertencentes aos serviços contemplados, em um único fluxo. Para isso, é necessário inserir no Fluxo de Transporte informações para que o decodificador MPEG-2 Sistemas saiba identificar a qual fluxo PES um determinado pacote pertence e, ainda, a qual serviço um dado fluxo PES pertence.

Cada fluxo PES possui um identificador de fluxo único, denominado stream_id. Para permitir ao decodificador MPEG-2 Sistemas identificar a qual fluxo PES um determinado pacote pertence, o padrão MPEG-2 Sistemas define que cada pacote de um fluxo PES, ao ser multiplexado em um Fluxo de Transporte, deve receber o identificador do fluxo PES a que pertence.

Uma vez que os fluxos PES não possuem nenhuma indicação sobre a qual serviço estão associados, faz-se necessário um mecanismo para a determinação de quais serviços estão presentes no Fluxo de Transporte e quais os fluxos PESs que compõem cada serviço. Esse mecanismo é especificado no MPEG-2 Sistemas através de um conjunto de tabelas, ou metadados, de informação específica de programa (PSI – *Program Specific Information*), conhecidas no contexto da TV digital como SI (*Service Information*). A SI denominada PMT (*Program Map Table*) contém a lista de identificadores dos fluxos PESs que compõem um serviço. As PMTs são localizadas através da SI denominada PAT (*Program Association Table*), que contém o identificador do fluxo elementar contendo as PMTs. O processo de identificação no decodificador, de cada serviço presente em um Fluxo de Transporte, é ilustrado na Figura 7.

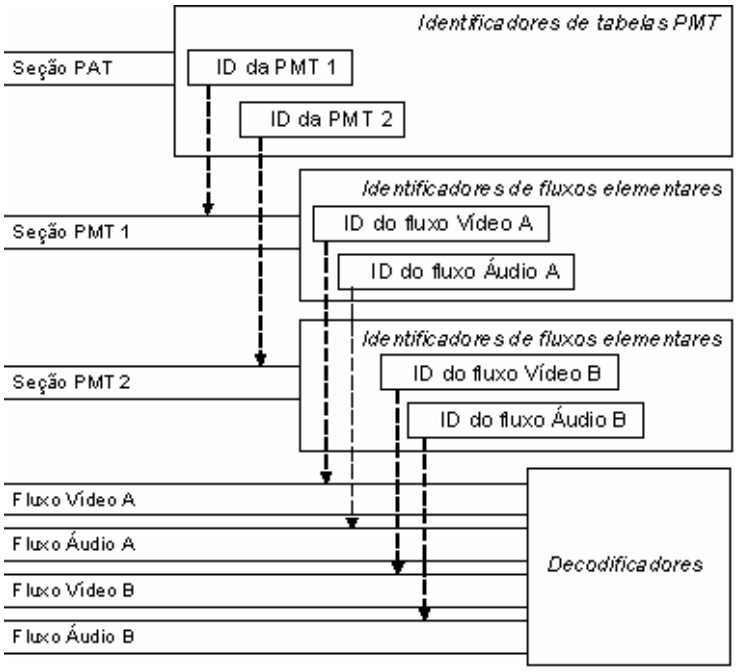


Figura 7: Relacionamento entre SIs e fluxos elementares.

O processo de identificação de cada serviço no decodificador inicia-se, quando do recebimento de um Fluxo de Transporte, pela busca da seção MPEG-2 que contém a SI PAT através de seu identificador conhecido. Segundo o padrão MPEG-2 Sistemas, as SIs PAT e PMT devem ser enviadas no Fluxo de Transporte a cada meio segundo, no máximo.

De posse das informações da SI PAT, são identificados os fluxos elementares que contêm as SIs PMT de cada serviço. Cada SI PMT fornece os identificadores de cada fluxo elementar que compõe o respectivo serviço. Os fluxos elementares referentes a cada serviço são, então, enviados para o módulo de decodificação apropriado.

Além das tabelas SIs e do conteúdo de áudio e vídeo principal, um Fluxo de Transporte MPEG-2 pode conter ainda outros dados, como arquivos, legendas, informações temporais, entre outros (ISO, 2000a).

O mecanismo de dividir cada fluxo elementar em pacotes, com o objetivo de possibilitar a multiplexação de fluxos elementares em um único Fluxo de Transporte, é ideal para fluxos elementares de áudio ou vídeo, onde cada pacote deve conter um certo número de quadros de vídeo ou amostras de áudio (ISO, 2000a). Entretanto, os fluxos de dados (incluindo as tabelas SIs) geralmente

possuem requisitos e características diferentes, que fazem com que essa divisão em pacotes torne-se mais complicada:

- Possivelmente existe mais de um item em um mesmo fluxo de dados como, por exemplo, diferentes tabelas SIs ou diferentes arquivos em uma transmissão de um sistema de arquivos. Assim, faz-se necessária uma maneira para determinar a forma com que os dados são organizados no fluxo;
- É possível existir mais de uma versão de um determinado tipo de dados. Por exemplo, as tabelas SIs ou módulos do carrossel de objetos, discutidos na próxima seção, podem ser atualizados e devem ser sobrescritos nos terminais de acesso;
- Existe a necessidade de identificar, ou mesmo oferecer suporte aos diferentes tipos de dados necessários.

Para atender a esses requisitos e características, o padrão MPEG-2 define o conceito de seções privadas, também chamadas apenas de seções, utilizadas para encapsular dados para serem multiplexados, de forma análoga aos pacotes PES.

As seções privadas MPEG-2 possuem a estrutura apresentada na Tabela 2. Elas são otimizadas para carregar dados DSM-CC, que serão discutidos a seguir, tabelas SIs, entre outros (ISO, 2000a). Para isso, possuem um cabeçalho especificado com o objetivo de informar ao decodificador como as seções estão sendo utilizadas para transportar os dados, como elas devem ser remontadas, qual o tipo de dados transportado, além de outros parâmetros para recuperação da informação (ISO, 2000a).

| Sintaxe |
|---|

```
private_section() {
       table_id
       section_syntax_indicator
       private_indicator
       Reserved
       private_section_length
       if(section_syntax_indicator == '0') {
              for(i=0; i<N; i++) {
                     private_data_byte
              }
       }
       else{
              table_id_extension
              Reserved
              version_number
              current_next_indicator
              section_number
              last_section_number
              for(i=0; i<private_section_length-9; i++) {
                     private_data_byte
              }
              CRC_32
       }
}
```

Tabela 10: Estrutura de uma Seção Privada MPEG-2 (ISO, 2000a)

## 3.2. DSM-CC

DSM-CC é uma extensão do padrão MPEG-2 Sistemas, publicado em 1996 como um padrão internacional ISO (ISO, 1998b). O DSM-CC foi desenvolvido para oferecer diversos tipos de serviços multimídia, entre eles a transmissão de dados multiplexados com o conteúdo audiovisual em um Fluxo de Transporte.

As especificações do padrão DSM-CC trazem definições interessantes que são comumente aplicadas ao contexto da TV digital. Entre as principais está o mecanismo de transmissão cíclica, denominado carrossel de objetos. Considerando que um usuário pode ligar seu terminal de acesso quando bem entender, é necessário aos provedores de conteúdo garantir a entrega da informação (dados como, por exemplo, imagens, texto, aplicações, entre outros (ISO, 1998b)) nos terminais de acesso.

### 3.2.1. Carrossel de Objetos DSM-CC

O carrossel de objetos DSM-CC tem como objetivo prover a transmissão cíclica de objetos. O padrão DSM-CC especifica três tipos de objetos: arquivos, diretórios e eventos. Assim, o carrossel de objetos pode possuir um verdadeiro

sistema de arquivos, isto é, um conjunto de diretórios e arquivos que, por exemplo, formam uma aplicação a ser executada nos terminais de acesso.

As especificações DSM-CC determinam que os dados transmitidos através do carrossel de objetos devem ser divididos em unidades denominadas módulos. Cada módulo pode possuir mais de um arquivo desde que não ultrapasse um total de 64 Kbytes. Os arquivos que estão em um mesmo módulo podem fazer parte de diretórios diferentes. Um arquivo que possui mais de 64 Kbytes deve ser transmitido em um único módulo, pois não é permitido dividir um arquivo em mais de um módulo.

Uma vez que os objetos foram dispostos em módulos, cada módulo é então transmitido, um após o outro. Após transmitir o último módulo, a transmissão é reiniciada desde o início (i.e. o primeiro módulo transmitido). O resultado disso é um fluxo elementar que contém o sistema de arquivos transmitido de forma cíclica. Assim, se um determinado terminal de acesso não recebeu uma parte de um módulo em particular (devido a um erro na transmissão ou por ter sido iniciado após a transmissão desse módulo), basta esperar pela retransmissão desse módulo.

Em alguns casos, utilizar o carrossel de objetos dessa forma significa introduzir retardos impraticáveis para os dados enviados pelo provedor de conteúdo. Por exemplo, para um carrossel que possui tamanho total de 172 Kbytes transmitidos a uma taxa de 128 Kbps, são necessários aproximadamente 11 segundos para a transmissão de um ciclo completo. Assim, nesse exemplo, existe um retardo máximo de 11 segundos para carregar um arquivo qualquer que faz parte desse carrossel. Para amenizar esse problema, os geradores de carrossel oferecem como opção transmitir alguns módulos com maior freqüência que outros. Assim, os módulos que contêm arquivos com maior prioridade podem ser transmitidos com maior freqüência.

Para que o exemplo seja mais concreto, considere a estrutura de diretórios da Figura 8. Essa estrutura conta com um arquivo `index.htm` que deve ser exibido nos terminais de acesso. O arquivo `index.htm` possui como âncora a imagem estática `lfgs.jpg`. Uma possível seleção dessa âncora, por parte do usuário do terminal de acesso, iniciaria a exibição dos arquivos `pagu.avi` e `chorinho.wav`.

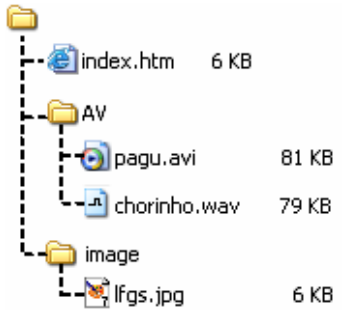


Figura 8: Estrutura de Diretórios.

Para transmitir a estrutura da Figura 8 via carrossel de objetos, primeiro é necessário dispor seus arquivos em módulos. Alocar o arquivo `index.htm` a um módulo é trivial. O próximo arquivo da estrutura (`pagu.avi`) possui mais de 64 Kbytes, não existindo a possibilidade de fazer parte do mesmo módulo de `index.htm`. Assim, `pagu.avi` é alocado em um módulo único. Analogamente, o arquivo `chorinho.wav` deve ser alocado em um terceiro módulo. Já o arquivo `lfgs.jpg` pode ser alocado no mesmo módulo que `index.htm`, uma vez que o tamanho desses dois arquivos somados é inferior a 64 Kbytes. A disposição final de arquivos em módulos é ilustrada na Figura 9.


Figura 9: Divisão da Estrutura de Diretórios da Figura 8 em Módulos.

O próximo passo é determinar a disposição dos módulos no carrossel. É interessante observar que se o carrossel possuir como ciclo de transmissão a seqüência de módulos: 1-2-3; um retardo de onze segundos poderia ser introduzido (para uma transmissão a 128 Kbps). Como os arquivos de áudio e vídeo dependem dos arquivos `index.htm` e `lfgs.jpg` para serem exibidos, uma disposição interessante de módulos do carrossel seria: 1-2-1-3, apresentada na Figura 10.

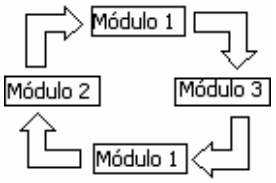


Figura 10: Disposição dos Módulos da Figura 9 no Carrossel de Objetos.

Finalmente, nesse exemplo, um objeto de evento deveria ser transmitido (nesse mesmo carrossel de objetos). Simplificadamente, o objeto de evento possui informações sobre os eventos DSM-CC que farão com que a aplicação responsável exiba o arquivo `index.htm`. Os objetos de evento, bem como os eventos DSM-CC, são explicados a seguir.

### 3.2.2. Eventos DSM-CC

Sincronizar o comportamento de uma determinada aplicação com o conteúdo de uma programação de TV específica é extremamente interessante, principalmente quando a aplicação em questão possui alguma relação semântica com essa programação. Por exemplo, considere como programação de TV um jogo de futebol onde existe uma aplicação, enviada anteriormente aos terminais de acesso através de carrossel de objetos, responsável por exibir o placar de outros jogos da rodada. Ao sair um gol em um desses outros jogos, é necessário que o placar seja atualizado na aplicação. Para esse tipo de sincronismo, é utilizada uma funcionalidade especificada no padrão DSM-CC, denominada eventos DSM-CC.

Um evento DSM-CC pode ser definido como uma ocorrência instantânea no tempo. Para criar um evento DSM-CC, é inserida uma estrutura no Fluxo de Transporte, denominada descritor de eventos. Cada descritor de eventos possui um identificador numérico para permitir que cada evento seja identificado de forma única no Fluxo de Transporte. Uma vez que não existe a possibilidade do provedor de conteúdo precisar exatamente onde o descritor é inserido no fluxo, cada descritor possui ainda uma referência temporal que indica em qual instante o evento deverá ocorrer. Como opção, um descritor de eventos pode informar ao sistema que o evento deve ocorrer imediatamente – esse tipo de evento é chamado de evento "*do it now*". Finalmente, o descritor de eventos possui ainda dados específicos das aplicações. No exemplo do parágrafo anterior, esses dados seriam responsáveis por informar qual time teria feito o gol, permitindo à aplicação realizar a alteração apropriada no placar.

Paralelamente à transmissão de um descritor de evento, o Fluxo de Transporte deve possuir ainda um carrossel de objetos contendo, pelo menos, um objeto de evento. Esse objeto é responsável por atribuir nomes aos eventos que podem ocorrer. Ou seja, o provedor de conteúdo envia por carrossel de objetos um

objeto de evento que possui uma tabela que relaciona nomes textuais, conhecidos pelas aplicações, aos identificadores numéricos dos eventos que serão criados pelos descritores. Um exemplo de objeto de evento é apresentado na Tabela 11. Note que cada nome textual deve ser associado a um único identificador. O nome textual presente na tabela do objeto de evento permite que cada aplicação se registre como *listener* de um tipo de evento DSM-CC. As aplicações podem ser preparadas para sempre se registrarem como *listeners* de eventos com nomes específicos. Por exemplo, a aplicação do placar discutida anteriormente é instruída para sempre se registrar em eventos com nome "Gol", "CartãoAmarelo" e "CartãoVermelho".

| Nome do Evento | ID do Evento |
|---|---|
| "Gol" | 04 |
| "Cartão Amarelo" | 06 |
| "CartãoVermelho" | 07 |

Tabela 11: Exemplo de Objeto de Evento DSM-CC.

Para tornar o exemplo mais concreto, é considerada a Figura 11. Um jogo entre Portuguesa e Marília é transmitido enquanto a aplicação exibe o placar, identificado pelo número 1 na figura, de outros jogos da rodada. Paralelamente, um carrossel de objetos contendo o objeto de evento apresentado na Tabela 11 é transmitido. No exemplo, a aplicação do placar se registra apenas para receber os eventos identificados com os valores numéricos 4, 6 ou 7, por possuírem os nomes "Gol", "Cartão Amarelo" e "CartãoVermelho", respectivamente. Ao ocorrer um gol em um dos outros jogos, o provedor de conteúdo envia o descritor de eventos, identificado por "a" na Figura 11, contendo as informações necessárias. Ao receber um evento cujo identificador numérico possui o valor 4, a aplicação reconhece que deve computar um gol de acordo com os dados presentes no descritor do evento e atualiza o placar (número 2 na Figura 11).

Analogamente, quando a aplicação recebe o evento, descrito em "b" na Figura 11, ela sabe que não deve computar um gol, mas sim informar que um cartão vermelho foi mostrado ao jogador "Nilmar" do time "Corinthians". A aplicação exibe assim o placar identificado pelo número 3 na Figura 11.

Conforme as especificações do padrão DSM-CC, descritores com mesmo identificador podem ser enviados quantas vezes forem necessários. Assim, ao

receber um segundo descritor cujo ID possui o valor numérico 4, a aplicação sabe que outro gol deve ser computado (descritor “c” e número 4 na Figura 11).

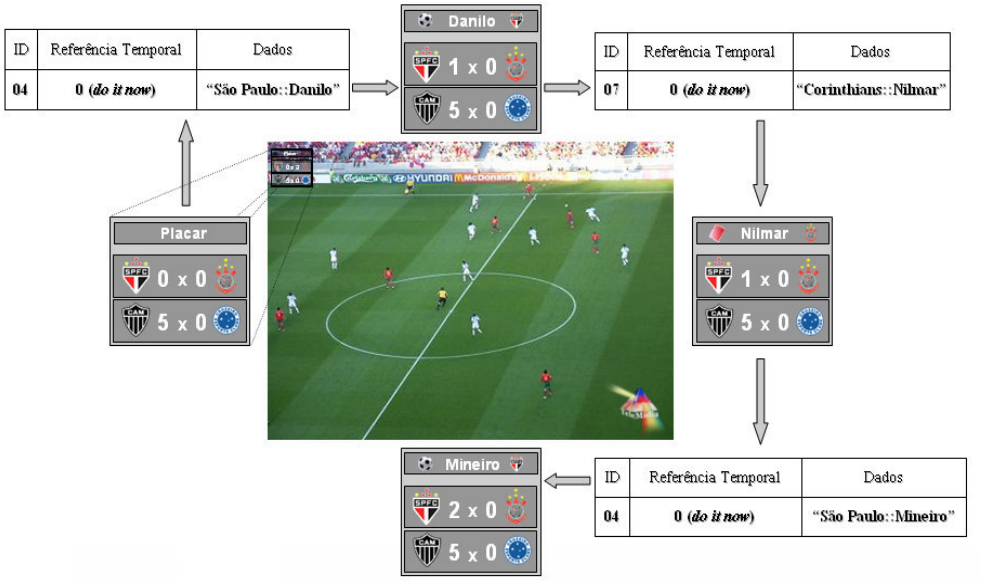


Figura 11: Exemplo de Sincronismo através de Eventos DSM-CC.

Através do exemplo discutido, pode-se observar que as aplicações para TV digital, na maioria das vezes, não seriam funcionais sem a possibilidade de manipular a apresentação de interfaces gráficas. O modelo de apresentação adotado na maioria dos sistemas de TV digital será discutido na próxima seção.

## 3.3. Modelo de Apresentação

Um modelo de apresentação de TV digital é comumente dividido em três camadas, ilustradas na Figura 12. A camada de fundo normalmente é capaz de exibir apenas uma cor, sendo que, em alguns casos, pode exibir uma imagem estática. Acima da camada de fundo, vem a camada de vídeo com capacidade de exibir o vídeo que está sendo decodificado por hardware (decodificadoras MPEG-2). A maioria dos terminais de acesso possui decodificadoras com recursos limitados, o que significa poucas opções de resolução como, por exemplo: a possibilidade do vídeo ser exibido ou em tela cheia ou em um quarto, ou um conjunto limitado de coordenadas para exibição desse vídeo.

Acima da camada de vídeo, vem a camada mais importante para as aplicações de TV digital, a camada gráfica. Todas as operações gráficas das aplicações são realizadas nessa camada. A camada gráfica pode possuir uma

resolução diferente das camadas de fundo e de vídeo, bem como apresentar um formato de *pixels* diferente. Geralmente, o formato dos *pixels* da camada de vídeo é retangular, enquanto que o formato dos *pixels* da camada gráfica possui uma base quadrada.

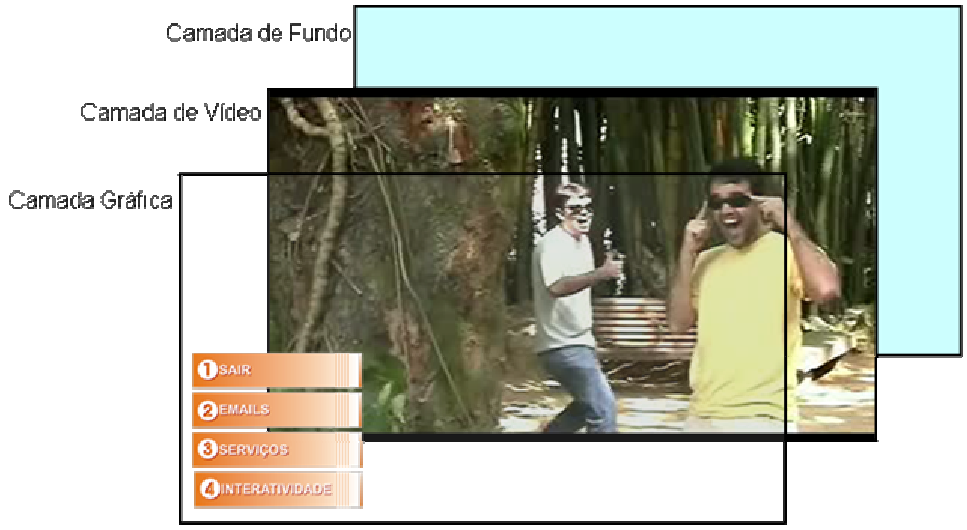


Figura 12: Modelo de Apresentação da TV Digital.

O modelo de apresentação é considerado como uma das partes mais complexas pelos desenvolvedores de middleware (Morris & Chaigneau, 2005). Ao mesmo tempo que as camadas devem ser desenvolvidas de forma independente, pois existem diferentes componentes nos terminais de acesso responsáveis pela geração de cada uma delas, é interessante também que haja algum tipo de interação entre elas. Assim, a forma específica com que uma camada é desenvolvida pode impor limitações às outras. Por exemplo, sempre que a camada de vídeo possuir o foco, a camada gráfica pode ficar impossibilitada de receber eventos do usuário.

Os principais middlewares existentes, como MHP e DASE, utilizam os componentes da ferramenta de janelas abstratas AWT (*Abstract Window Toolkit*), seguindo as especificações HAVi (*Home Audio/Video Interoperability*) (HAVi, 1999) para controlar a imagem, ou a cor, do plano de fundo. Além disso, os componentes HAVi são utilizados na camada gráfica para renderizar as interfaces gráficas das aplicações e exibir vídeos secundários (Morris & Chaigneau, 2005). O vídeo decodificado por hardware é exibido diretamente na camada de vídeo.

Os componentes HAVi/AWT, entretanto, dependem de uma máquina virtual Java. Isso significa, conforme discutido no Capítulo 1, perda no

desempenho e custos sobre royalties e propriedade intelectual. Para alcançar melhor desempenho e utilizar o mínimo de recursos, este trabalho considera a biblioteca DirectFB (Hundt, 2004). Resumidamente, DirectFB oferece uma abstração de dispositivos, através do sistema operacional. Assim, antes de discorrer sobre a biblioteca DirectFB, os sistemas operacionais serão discutidos na próxima seção.

## 3.4. Sistemas Operacionais para Terminais de Acesso

Os sistemas operacionais voltados para computadores em geral oferecem um conjunto de chamadas de sistema e bibliotecas de programação que possibilitam a criação de processos (representação interna de programas em um sistema), o gerenciamento de memória, o acesso a sistemas de arquivos e a comunicação básica entre processos (Tanenbaum, 1992). Simplificadamente, pode-se dizer que sistemas operacionais gerenciam o ambiente de execução de programas, definindo abstrações para tornar mais amigável e singular o acesso a estruturas de controle de mais baixo nível, relacionadas diretamente com o hardware. Foram concebidos no intuito de tornar mais simples a construção de aplicações, além de permitir a concorrência de execução entre tais aplicações, permitindo assim um compartilhamento no uso dos recursos.

Por sua vez, terminais de acesso de TV digital possuem hardware especializado para a exibição de conteúdo televisivo e forte limitação na quantidade de recursos computacionais. Em TV digital, vários outros requisitos vêm à tona, como aqueles impostos pela recepção de dados por difusão, sincronismo, interação por controle remoto etc. Além desses, os recursos controlados por um sistema operacional para terminais de acesso incluem comunicação em rede (por meio de um canal de retorno), gerenciamento de energia, dispositivos de som e dispositivos gráficos.

Nota-se que alguns desses recursos podem ser críticos para a execução de uma aplicação multimídia distribuída, como assim o são para vários tipos de programas interativos, que podem ser disponibilizados no contexto de um sistema de TV digital. Um gerenciamento adequado, respeitando os requisitos de desempenho sobre esses recursos críticos (e.g. CPU e canal de retorno), é fundamental em qualquer perfil de terminal de acesso.

De qualquer forma, todas as funcionalidades, até aqui descritas, presentes em um sistema operacional, devem estar acessíveis para os demais componentes de software, por meio de uma API. Essa API consiste na interface oferecida pelos sistemas operacionais de terminais de acesso, com o objetivo de prover as funções para a comunicação entre o software em execução e o hardware, assim como as demais abstrações comumente definidas internamente em um sistema operacional convencional. Uma alternativa para oferecer uma abstração para a construção e manipulação do modelo de apresentação sobre o sistema operacional Linux[6] é dada pela biblioteca DirectFB. Na verdade, além de oferecer uma abstração ao modelo de apresentação, a biblioteca DirectFB oferece ainda abstrações sobre outros dispositivos (controle remoto, teclado, mouse etc.). DirectFB é o assunto da próxima seção.

## 3.5. DirectFB

DirectFB é uma biblioteca leve, disponível como código aberto, que oferece aceleração gráfica através de um dispositivo do sistema operacional Linux denominado *FrameBuffer Device* (Uytterhoeven, 2001). O *FrameBuffer* é um recurso nativo do *kernel* do Linux que foi originalmente desenvolvido para oferecer a visualização de imagens gráficas enquanto o sistema estivesse operando em modo texto. Assim, DirectFB permite a manipulação de interfaces gráficas sem a necessidade da existência de um servidor X (XFree, 1994).

Além da aceleração gráfica, o tamanho reduzido é uma outra importante vantagem no uso de DirectFB: uma instalação completa, com os *drivers* para aceleração gráfica e fontes, pode ocupar menos de 15 MB, enquanto uma instalação completa do Xfree86 4.2 (XFree, 1994) ocupa geralmente mais de 100 MB. Além disso, DirectFB é compatível com mais de 90% das placas de vídeo convencionais (Hundt, 2004), oferecendo ainda uma abstração para dispositivos de entrada.

---

[6] Linux foi escolhido por ser um sistema operacional gratuito, de código aberto e amplamente utilizado em terminais de acesso para TV digital . Além disso, Linux foi definido como o sistema operacional do modelo de referência proposto para o SBTVD, o qual consiste em um dos principais focos definidos para este trabalho.

O DirectFB foi desenvolvido através de interfaces. No contexto do DirectFB, uma interface é uma estrutura da linguagem C que contém ponteiros para funções ou para outras interfaces. O diagrama de interfaces do DirectFB é apresentado na Figura 13.

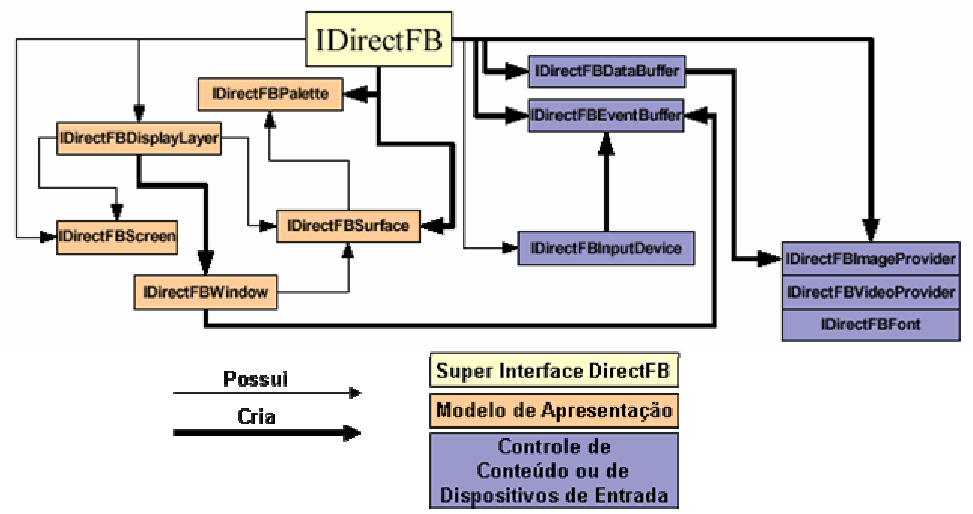


Figura 13: Diagrama de Interfaces do DirectFB

Na Figura 13, as interfaces em laranja (*IDirectFBLayer*, *IDirectFBScreen*, *IDirectFBWindow*, *IDirectFBSurface* e *IDirectFBPalette*) possuem funções para criação e manipulação do modelo de apresentação. Já as interfaces em azul (*IDirectFBInputDevice*, *IDirectFBDataBuffer*, *IDirectFBEventBuffer*, *IDirectFBImageProvider*, *IDirectFBVideoProvider* e *IDirectFBFont*) possuem funções para controlar dispositivos de entrada (*IDirectFBInputDevice*, *IDirectFBDataBuffer* e *IDirectFBEventBuffer*) ou para controlar o conteúdo das mídias exibidas no modelo de apresentação (*IDirectFBImageProvider*, *IDirectFBVideoProvider* e *IDirectFBFont*). Todas as interfaces podem ser obtidas através da super interface em amarelo *IDirectFB*, a única a ser obtida através de uma função global (*DirectFBCreate()*).

A Figura 14 ilustra como as interfaces definidas em DirectFB criam um modelo de apresentação para TV digital, conforme discutido na Seção 2.3. As camadas do modelo de apresentação são representadas pela interface *IDirectFBDisplayLayer*. Cada *IDirectDisplayFBLayer* possui informações para configurar seus recursos. Por exemplo, a interface que representa a camada de fundo, do modelo de apresentação, possui apenas atributos para configurar uma cor de fundo ou, no máximo (dependendo dos recursos da plataforma), uma

imagem de fundo. A interface que representa a camada de vídeo possui atributos para apresentar uma janela capaz de renderizar um vídeo. A interface que representa a camada gráfica, por sua vez, pode (dependendo da plataforma) ter atributos para apresentar janelas capazes de renderizar textos, imagens e vídeos.

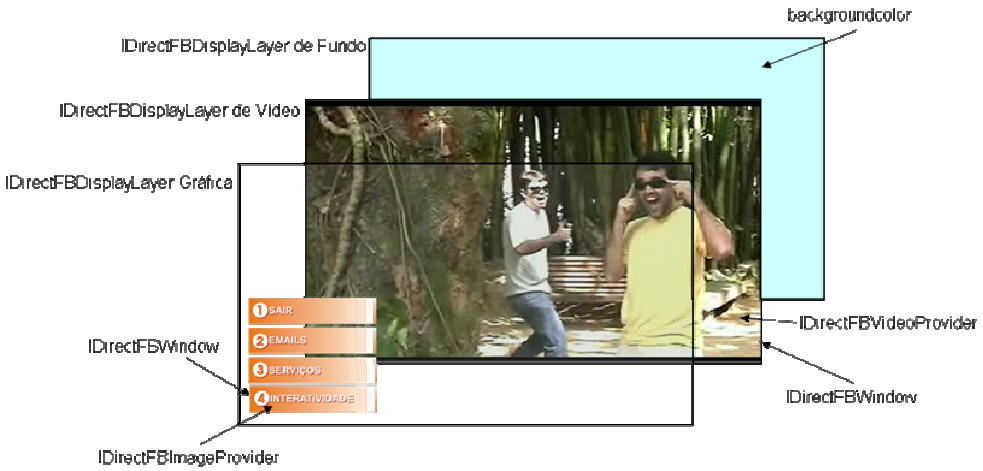


Figura 14: Modelo de Apresentação sob a Ótica do DirectFB

As janelas, criadas através da interface *IDirectFBWindow*, possuem uma superfície (*IDirectFBSurface*) capaz de renderizar o conteúdo de mídias decodificadas. Para decodificação, é necessário associar à superfície uma interface que trate o tipo específico de mídia. As interfaces disponíveis no DirectFB para essa finalidade são: *IDirectFBImageProvider*, *IDirectFBVideoProvider* e *IDirectFBFont*. As interfaces *IDirectFBImageProvider* e *IDirectFBVideoProvider* controlam a decodificação de imagens estáticas e vídeos, respectivamente, através de bibliotecas específicas. A interface *IDirectFBFont* permite exibir dados textuais.

Informações sobre a capacidade de uma superfície, como por exemplo, resoluções e cores disponíveis, se possui capacidade de renderizar vídeos, entre outras (Hundt, 2004), podem ser obtidas através da interface *IDirectFBPalette*. De forma análoga, a interface *IDirectFBScreen* provê informações sobre as camadas de uma plataforma, tais como, número de camadas suportadas pela plataforma, capacidades de apresentação de cada camada, entre outras (Hundt, 2004). Finalmente, cada janela é capaz de receber eventos (*IDirectFBEventBuffer*) através de dispositivos de entrada. A interface *IDirectFBInputDevice* oferece uma abstração desses dispositivos.

É importante ressaltar que toda renderização gráfica realizada pelas interfaces do DirectFB utilizam o dispositivo *FrameBuffer*, fazendo com que a manipulação de interfaces gráficas seja uma tarefa com baixo custo de processamento, quando comparado ao de outras bibliotecas que não utilizam esse dispositivo como, por exemplo, *XFree* e AWT.

# 4
# Arquitetura do Middleware *Maestro*

O middleware declarativo apresentado neste trabalho, denominado *Maestro*, faz parte da proposta para o padrão de sincronismo de mídias do modelo de referência do Sistema Brasileiro de TV Digital. A arquitetura desse middleware foi elaborada de forma modular, tendo como módulo central, ou núcleo, uma reestruturação, direcionada à TV digital, do Formatador HyperProp (Rodrigues, 2003). As seções seguintes descrevem a arquitetura modular do *Maestro*, bem como as APIs oferecidas por seus principais módulos.

## 4.1.
## Arquitetura Modular

O middleware declarativo *Maestro* possui a arquitetura modular ilustrada pela Figura 15. Entre os principais módulos, que fazem parte dessa arquitetura, estão: Sintonizador, Filtro de Seções, DSM-CC, Núcleo e Exibidores. As próximas seções descrevem cada um dos módulos. Uma atenção especial será dada a um sub-módulo específico do Núcleo, denominado módulo Gerenciador de Bases Privadas, por ser uma contribuição que estende o subconjunto do Formatador HyperProp (Rodrigues, 2003), definido como Núcleo do middleware *Maestro*.

É interessante ressaltar que as APIs dos módulos *Sintonizador*, *Filtro de Seções* e *DSM-CC* são baseadas nas APIs definidas no padrão DVB (DVB, 2004). Os sub-módulos Filtro de Seções, Sintonizador, DSM-CC, Gerenciador de Bases Privadas e Gerenciador DSM-CC, apresentados na Figura 15, foram modelados de acordo com as discussões realizadas, mas suas implementações são citadas entre os trabalhos futuros definidos nesta dissertação.

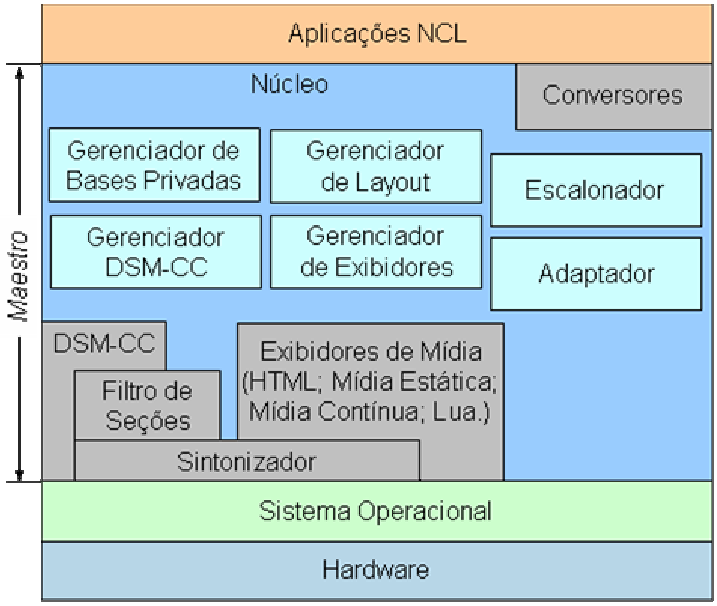


Figura 15: Arquitetura Modular do Middleware *Maestro*.

## 4.2. Módulo Sintonizador

A justificativa para a definição de um módulo sintonizador, na arquitetura do middleware *Maestro*, está na forma com que é realizada a transmissão dos fluxos de transporte. No contexto da TV digital, cada fluxo de transporte é enviado em um canal distinto, através de um determinado sistema de transmissão por difusão, que pode ser um sistema de difusão terrestre, por satélite ou por cabo (Morris & Chaigneau, 2005). Os canais podem ser multiplexados na freqüência, ou no tempo, como, por exemplo, em redes IP, onde os canais podem ser determinados por um endereço IP *multicast* (Kurose & Ross, 2003), somado ao número de uma porta lógica para a recepção do fluxo.

Para sintonizar um canal, com objetivo de receber um fluxo de transporte, primeiro é necessário definir entre as interfaces de redes, presentes em um terminal de acesso, quais estão aptas a receber fluxos de transporte. Por exemplo, um terminal de acesso que possui uma interface de rede receptora de sinais de TV terrestre e outra receptora de TV por satélite. Dessa forma, é especificado, no módulo sintonizador, um gerenciador de interfaces de rede, denominado *NetworkInterfaceManager*, responsável por gerenciar todas as instâncias de interface de rede. Cada instância dessas interfaces é denominada *NetworkInterface*. O gerenciador de interfaces de rede oferece a API apresentada

na Tabela 12. Note, na Tabela 12, que o gerenciador de interfaces é definido utilizando o padrão de projeto *Singleton* (Gamma et al, 1995), possuindo uma instância única na arquitetura, que pode ser obtida através do método *getInstance()*.

| Método | Descrição |
|---|---|
| getInstance() | Retorna a instância única do gerenciador, uma vez que o mesmo é definido como *Singleton* (Gamma et al, 1995). |
| getNetworkInterfaces() | Retorna todas as interfaces de rede presentes no terminal de acesso. |
| setCurrentNetworkInterface( NetworkInterface) | Define qual interface de rede, passada como parâmetro, pode ser utilizada para sintonizar um fluxo de transporte. |
| getCurrentNetworkInterface() | Retorna a interface de rede definida pelo método setCurrentNetworkInterface(). |

Tabela 12: API oferecida pelo *NetworkInterfaceManager*.

Cada instância *NetworkInterface* encontrada pelo gerenciador de interfaces de rede representa uma forma possível de se obter fluxos de transporte. Geralmente, para terminais que apresentam mais de uma interface de rede, o usuário seleciona, através do controle remoto, qual das interfaces de rede deve ser utilizada para sintonizar os canais. Para isso foi especificado o método *setCurrentNetworkInterface()*. No entanto, para que o gerenciador de interfaces ofereça informações ao usuário de sua API sobre quais interfaces de rede estão aptas a sintonizar fluxos de transporte, deve-se percorrer todas as instâncias de interfaces de rede, obtidas através do método *getNetworkInterfaces()*. Essa verificação pode ser realizada, por exemplo, rastreando os intervalos de freqüências (para redes de TV por difusão terrestre, cabo e satélite); ou endereços IPs somados às portas lógicas (em um sistema de TV sobre IP) que cada interface alcança.

A API de uma *NetworkInterface*, apresentada na Tabela 13, oferece o método *listAccessibleTransportStreams()*. Esse método realiza suas tarefas, apresentadas na Tabela 13, apenas depois de verificar que o valor retornado pelo método *isReserved()* é falso. Isso significa que nenhum usuário está, concomitantemente, utilizando a interface de rede para sintonizar um canal específico.

Para obter o fluxo de transporte relativo ao canal que a interface de rede está atualmente sintonizada, deve-se utilizar o método *getCurrentTransportStream()*. Os usuários da API de uma *NetworkInterface* específica, que manipulam constantemente o fluxo de transporte retornado por esse método (por exemplo, módulo Filtro de Seções), devem registrar-se como *listener* dessa interface de rede. Dessa forma, esses usuários recebem notificações quanto à ocorrência de uma nova sintonização (i.e. deve ser considerado um novo fluxo de transporte a ser manipulado).

Para obter informações sobre o tipo de sistema de transmissão que a interface de rede suporta, deve-se utilizar o método *getDeliverySystemType()*.

| Método | Descrição |
|---|---|
| listAccessibleTransportStreams() | Retorna uma lista vazia se o valor do método isReserved() retornar verdadeiro. Caso contrário, realiza um rastreamento nos canais (intervalo de freqüências para redes de TV por difusão terrestre, cabo e satélite; ou endereços IPs somados às portas lógicas pré-determinadas em um sistema de TV sobre IP) que a interface alcança e retorna uma tabela, relacionando, para cada fluxo de transporte acessível, o canal correspondente. |
| getCurrentTransportStream() | Retorna o fluxo de transporte correspondente ao canal atualmente sintonizado. Retorna um valor nulo se nenhum canal válido estiver sintonizado. |
| getDeliverySystemType() | Retorna um valor inteiro que identifica o tipo de sistema de transmissão que a interface de rede suporta. Os sistemas de transmissão possíveis são: transmissão de TV a cabo, TV por satélite ou TV terrestre, bem como um sistema de transmissão de TV sobre IP *multicast*. |
| isReserved() | Informa se a interface está ou não reservada. Interfaces de rede são reservadas via um controlador de interface de rede, conforme explicado na Tabela 14. |
| addNIListener(NIListener) | Adiciona um *listener*, passado como parâmetro, à interface de rede. Os *listeners* são notificados que uma nova sintonização ocorreu e recebem uma referência para o recurso da interface de rede por onde o fluxo de transporte está sendo recebido. |
| removeNIListener(NIListener) | Remove um *listener*, passado como parâmetro, da interface de rede. |

Tabela 13: API oferecida pelo *NetworkInterface*.

Para controlar uma interface de rede, de forma a utilizá-la para sintonizar um canal, com o objetivo de receber um fluxo de transporte, faz-se necessário um controlador de interface de rede, denominado *NetworkInterfaceController*. A API oferecida pelo controlador de interface é apresentada na Tabela 14.

| Método | Descrição |
|---|---|
| reserve() | Realiza uma associação entre a interface de rede definida no gerenciador de interfaces de rede, através do método *setCurrentNetworkInterface()*, e o controlador de interface de rede, reservando a interface de rede para realizar uma sintonização. Essa operação é realizada apenas se o método isReserved() da interface definida retornar falso. |
| tune(int) | Utiliza a interface de rede associada ao controlador, através do método reserve(), para sintonizar o fluxo de transporte correspondente ao canal passado como parâmetro. Retorna uma referência para o fluxo de transporte sintonizado. |
| release() | Desfaz a associação com a interface de rede, liberando seus recursos para possibilitar uma futura sintonização. |
| getNetworkInterface() | Retorna a interface de rede associada ao controlador. |

Tabela 14: API oferecida pelo *NetworkInterfaceController*.

Após criar um controlador de interface de rede, é necessário associá-lo à interface desejada. Essa associação é realizada através do método *reserve*() descrito na Tabela 14, para depois sintonizar o fluxo de transporte desejado, com o método *tune*(). Uma vez sintonizado, o fluxo de transporte é recebido através de um recurso da interface de rede.

## 4.3. Módulo Filtro de Seções

Como discutido no Capítulo 2, as seções privadas MPEG-2 são utilizadas para transportar dados que variam de tabelas SIs a sistemas de arquivos. Um middleware de TV digital deve permitir que partes específicas de um fluxo de transporte, relativas a uma seção, sejam obtidas. Para isso, o middleware *Maestro* define um filtro, denominado *SectionFilter*, que oferece uma API capaz de retornar seções privadas contidas no fluxo de transporte MPEG-2 sintonizado, de acordo com alguns critérios especificados como, por exemplo, “filtre as seções cujo tipo de dados são tabelas SIs”. Uma vez que um *SectionFilter* realiza processamento sobre o fluxo de transporte sintonizado, é necessário que o mesmo

se cadastre como *listener* da *NetworkInterface*, discutida na seção anterior, que possui o fluxo sintonizado.

Segundo a arquitetura do middleware *Maestro*, ilustrada na Figura 15, a API do módulo Filtro de Seções é utilizada pelos módulos DSM-CC e Núcleo. O módulo DSM-CC normalmente solicita as seções contendo carrossel de objetos ou eventos DSM-CC. O Núcleo, por sua vez, solicita as seções contendo um determinado tipo de tabela SI, como será discutido na Seção 4.5.

A API de um *SectionFilter*, apresentada na Tabela 15, permite que restrições afirmativas e restrições de negação sejam especificadas no filtro para designar as seções desejadas. Por exemplo, um usuário da API *SectionFilter* que deseja receber apenas carrossel de objetos que não possua aplicações NCL especificaria o seguinte filtro: "filtre todas as seções com dados de um carrossel de objetos (`table_id` = 10, no exemplo), mas que não contenha aplicações NCL (`table_id_extension` ≠ 32, no exemplo)".

| Método | Descrição |
|---|---|
| startFiltering(<br>Pid,<br>table_id,<br>posFilerMask,<br>negFilerMask); | Inicia o processamento do fluxo de transporte sintonizado, retornando uma referência para as seções encontradas de acordo com o filtro especificado. Os parâmetros são opcionais. Pid especifica o identificador de fluxo, table_id especifica o identificador da seção, posFileMask são restrições afirmativas e negFilerMask definem restrições de negação. |
| setTimeOut(Time) | Especifica um tempo limite para que um temporizador, passado como parâmetro, pare o processamento do filtro no fluxo de transporte. Sempre que uma seção for encontrada, o temporizador é reiniciado. |
| stopFiltering() | Pára o processamento do filtro sobre o fluxo de transporte. |

Tabela 15: API oferecida pelo *SectionFilter*.

## 4.4. Módulo DSM-CC

Para oferecer informações e conteúdo interativo em conjunto com a programação audiovisual, um provedor de conteúdo deve enviar aplicações (documentos NCL, por exemplo) aos terminais de acesso. Um dos mecanismos utilizados para que essas aplicações cheguem aos terminais de acesso é o mecanismo de transmissão cíclica, especificamente o carrossel de objetos DSM-CC. Nesse caso, o módulo Filtro de Seções é utilizado para entregar o fluxo de

dados ao módulo DSM-CC, responsável por tratar as funcionalidades desse protocolo. Uma das principais funções do módulo DSM-CC consiste em decodificar o carrossel de objetos, criando um sistema de arquivos para esse carrossel, de acordo com o sistema operacional da plataforma. Uma outra importante função do módulo DSM-CC é monitorar a chegada de eventos de fluxo (*stream events*) e notificar aqueles usuários que tenham se registrado como *listeners* desses eventos.

O módulo DSM-CC foi definido na arquitetura do middleware *Maestro* com o objetivo de oferecer APIs para manipular o carrossel de objetos, bem como eventos de fluxo, obtidos através do módulo Filtro de Seções.

O padrão DSM-CC (ISO, 1998b) define o conceito de "*service domain*" como um conjunto de objetos, pertencentes a um carrossel de objetos, que formam um sistema de arquivos. Para representar um conjunto de objetos, o módulo DSM-CC define um *ServiceDomain*, que oferece a API apresentada pela Tabela 16.

| Método | Descrição |
|---|---|
| attach(Carousel) | Realiza uma associação do *ServiceDomain* a um carrossel de objetos, passado como parâmetro, montando esse carrossel em um diretório do sistema de arquivos. |
| attach(Locator, Id) | Recebe como parâmetro uma referência para um serviço do fluxo de transporte sintonizado, bem como o ID de um carrossel de objetos específico, presente nesse serviço. Para encontrar esse carrossel de objetos, a API do módulo Filtro de Seções é utilizada. Depois, realiza uma associação do *ServiceDomain* ao carrossel de objetos, retornado pelo módulo Filtro de Seções, montando esse carrossel em um diretório do sistema de arquivos. |
| getMountPoint() | Retorna o diretório onde o carrossel de objetos foi montado. |
| detach() | Realiza um *unmount* no sistema de arquivos, removendo os arquivos criados, e desfazendo a associação realizada por qualquer um dos métodos *attach*. |
| addEventListener( ObjectEventListener) | Adiciona o *listener* passado como parâmetro, para ser notificado sobre eventos gerados pelo carrossel de objetos. |
| removeEventListener( ObjectEventListener) | Remove o *listener* passado como parâmetro. |

Tabela 16: API oferecida pelo *ServiceDomain*.

Um *ServiceDomain* deve ser associado a um carrossel de objetos, através do método *attach()* (qualquer uma das duas opções), apresentado na Tabela 16. Esse método é responsável por montar (semelhante ao comando `mount` do sistema

operacional UNIX (Tanenbaum, 1992)), em um diretório do sistema operacional do terminal de acesso, o sistema de arquivos representado pelos objetos do carrossel de objetos associado. Note que nenhum dos métodos apresentados na Tabela 16 permite especificar em qual diretório um carrossel deve ser montado. Isso é feito para evitar conflitos de diretórios definidos por usuários da API. Assim o método *attach()* define internamente onde montar o carrossel de objetos. O local definido pelo método *attach()* pode ser obtido através do método *getMountPoint()*.

Para ser notificado que um objeto contido no carrossel de objetos foi completamente recebido, um usuário da API *ServiceDomain* (módulo Núcleo, conforme ilustra a Figura 15) pode se cadastrar como um *listener* de eventos. Na notificação é passada como parâmetro a localização, no sistema de arquivos, desse objeto.

Um dos recursos mais utilizados do carrossel de objetos é a atualização de arquivos, ou seja, o provedor de conteúdo pode enviar novas versões de módulos no carrossel (Seção 3.2.1). Cada atualização gera também uma notificação aos *listeners* de eventos.

Após utilizar os objetos de um carrossel de objetos, o usuário da API pode removê-los através do método *detach()*. O método *detach()* pode ser chamado também quando um novo fluxo de transporte for sintonizado.

Conforme discutido no Capítulo 2, além de arquivos e diretórios, um carrossel de objetos pode possuir objetos de evento, responsáveis por relacionar nomes textuais, conhecidos pelas aplicações, aos identificadores numéricos dos eventos que serão criados pelos descritores. Quando um objeto de evento é recebido, *ServiceDomain* notifica seus *listeners*, passando esse objeto como referência. Para representar um objeto de evento foi definido um *StreamEventObject*, que oferece a API apresentada na Tabela 17.

| Método | Descrição |
|---|---|
| subscribe( eventName, StreamEventListener) | Cadastra o *listener* passado como parâmetro, com o objetivo de ser notificado sobre ocorrências de eventos do tipo eventName, também passado como parâmetro. Retorna o identificador do tipo de evento solicitado. |
| unsubscribe( eventId, StreamEventListener) | Desfaz o cadastro do *listener* passado como parâmetro no tipo de evento correspondente a um identificador, também passado como parâmetro. |
| getEventList() | Retorna uma tabela relacionando identificadores aos tipos de eventos que podem ocorrer. |

Tabela 17: API oferecida pelo *StreamEventObject*.

Um usuário da API oferecida pelo módulo DSM-CC pode cadastrar-se para receber um determinado tipo de evento DSM-CC através do método *subscribe()*. Para saber quais os tipos de eventos estão disponíveis para cadastro, é utilizado o método *getEventList()*. Esse método retorna uma tabela relacionando tipos de eventos aos seus identificadores, conforme discutido no Capítulo 2.

Além do carrossel de objetos, eventos DSM-CC podem ser obtidos através do módulo Filtro de Seções. Os eventos DSM-CC, especificados por um descritor de eventos, são representados no módulo DSM-CC do middleware *Maestro* por um *StreamEvent*. A API oferecida por um *StreamEvent*, apresentada na Tabela 18, permite acesso aos principais campos de um evento DSM-CC.

| Método | Descrição |
|---|---|
| getEventName() | Retorna o nome do evento, utilizado para definir seu tipo. |
| getEventId() | Retorna o identificador do evento. |
| getEventTimeReference() | Retorna a referência temporal para a ocorrência do evento. Retorna um valor negativo caso o evento seja um evento "*do it now*". |
| getEventData | Retorna os dados específicos da aplicação. |

Tabela 18: API oferecida pelo *StreamEvent*.

Quando um evento é obtido pelo módulo DSM-CC, sua referência temporal é verificada com o objetivo de agendar sua ocorrência. Para um evento "*do it now*", imediatamente é utilizado o método *getEventName()* para que todos os *listeners* cadastrados para receber esse tipo de evento sejam notificados. Para um evento com uma referência temporal ainda válida (se for uma referência temporal passada, o evento é descartado), um monitor é instanciado para disparar sua

ocorrência no instante apropriado. A notificação da ocorrência no instante apropriado é análoga à notificação dos eventos "*do it now*".

## 4.5. Núcleo e Exibidores

O núcleo do middleware *Maestro* consiste em um formatador responsável por controlar a apresentação de documentos NCL. Os principais sub-módulos que compõem o Núcleo foram apresentados na Figura 15, página 66.

Através de um conversor, um documento NCL recebido pelo middleware *Maestro* é traduzido para uma estrutura de execução apropriada para o controle da apresentação dos documentos. O modelo de execução proposto para o middleware *Maestro* é baseado no modelo de contextos aninhados (NCM) (Soares et al, 2003).

Entre as principais entidades do modelo NCM estão: nós de composição, nós de mídia (ou objetos de mídia), descritores, âncoras, eventos e elos. Simplificadamente, as *composições* permitem estruturar os documentos e os *nós de mídia* representam os objetos cujos conteúdos são unidades de informações em uma determinada mídia a serem exibidos. Uma composição NCM é formada por uma coleção de nós e elos (explicados mais adiante), podendo esses nós serem objetos de mídia ou outras composições. *Descritores* podem ser associados a objetos de mídia, indicando como esses objetos devem ser apresentados. *Âncoras* representam um conjunto de unidades de informação marcadas de um nó, ou um atributo do nó e permitem definir eventos. Os *elos* modelam as relações entre os eventos. Finalmente, *eventos* são ocorrências no tempo que podem ser instantâneas ou de duração finita. Em cada objeto de mídia, vários eventos podem ser estabelecidos, como a apresentação de uma âncora (exibição de um trecho de um vídeo, por exemplo), a seleção de uma âncora pelo usuário telespectador etc. (Soares et al, 2003).

No middleware *Maestro* cada documento NCL é convertido para uma composição NCM, chamada de *contexto*. As entidades de um documento NCL são diretamente convertidas para entidades do modelo NCM, uma vez que, como comentado no Capítulo 1, a linguagem teve por base o Modelo de Contextos Aninhados.

Com o objetivo de agrupar um conjunto de documentos NCL, é utilizado o conceito (também definido no NCM) de um tipo especial de composição denominada *base privada*. Cada vez que um fluxo de transporte é sintonizado, o middleware abre uma nova base privada para agrupar os documentos transmitidos. A manutenção desses agrupamentos é a função atribuída ao sub-módulo Gerenciador de Bases Privadas. Esse sub-módulo utiliza a API do módulo Filtro de Seções para identificar cada base privada com o identificador do fluxo de transporte[7] sintonizado. Além disso, esse sub-módulo registra-se como *listener* do módulo sintonizador. Assim, a cada notificação recebida (sobre uma nova sintonização), o sub-módulo Gerenciador de Bases Privadas exclui a base privada que estava sendo utilizada e cria uma nova, através do identificador do novo fluxo de transporte sintonizado. Isso significa que é utilizada apenas uma base privada por vez. Vale ressaltar que, antes de sintonizar um novo canal, é possível salvar a base privada, que está sendo utilizada, em um repositório. Esse recurso, evidentemente, irá depender da capacidade do terminal de acesso.

As manipulações sobre bases privadas são oferecidas pelo sub-módulo Gerenciador de Bases Privadas através da API de um *PrivateBaseManager*, apresentada na Tabela 19. Através das bases privadas, os documentos NCL, que estão sendo apresentados (ou já foram apresentados), podem ser organizados.

| Método | Descrição |
|---|---|
| openBase (baseId) | Abre uma base privada existente ou cria uma nova base, caso ainda não exista. |
| saveBase (baseId) | Salva conteúdo da base privada. |
| closeBase (baseId) | Fecha a base privada especificada. |
| deleteBase(baseId) | Exclui a base privada especificada. |

Tabela 19: API oferecida pelo *PrivateBaseManager*.

O sub-módulo Gerenciador DSM-CC, ilustrado na Figura 15, permite, entre outras funcionalidades, que os provedores de conteúdo acessem as bases privadas, iniciando, ou mesmo organizando, suas aplicações NCL, nos terminais de acesso. Esse sub-módulo recebe, através do módulo DSM-CC, todos os eventos DSM-CC gerados pelo provedor de conteúdo. Esses eventos são então interpretados e

[7] Normalmente, os fluxos de transporte possuem uma identificação fixa e única para cada provedor de conteúdo.

mapeados para chamadas aos métodos de outros sub-módulos do middleware *Maestro*.

A apresentação de um documento NCL transmitido tem seu início quando o sub-módulo Gerenciador DSM-CC recebe um evento DSM-CC que transporta, no campo "nome do evento", o valor "*prepareNCL*", e no campo "dados específico das aplicações", as seguintes informações: uma referência para um carrossel de objetos (serviço do fluxo de transporte sintonizado e o ID do carrossel) que está sendo transmitido; uma referência para o documento NCL presente nesse carrossel; um identificador opcional da base privada onde as entidades do documento serão dispostas.

A partir do recebimento desse evento, o sub-módulo Gerenciador DSM-CC utiliza a API do *ServiceDomain* para montar o carrossel no sistema de arquivos, registrando-se como *listener* desse *ServiceDomain* (para receber notificações sobre atualizações nos módulos, por exemplo). Normalmente, quando esse evento DSM-CC chega ao terminal de acesso, o carrossel de objetos já está disponível, em um dispositivo (*device* - /dev do sistema operacional) do terminal de acesso, para ser montado no sistema de arquivos. Caso não esteja disponível, o sub-módulo Gerenciador DSM-CC deve aguardar uma notificação da disponibilidade por parte do *ServiceDomain*.

Após ser montado o sistema de arquivos, correspondente ao carrossel de objetos especificado, o sub-módulo Gerenciador DSM-CC solicita o serviço de um compilador NCL, presente no módulo Conversores. Essa chamada entrega como parâmetro a referência do documento NCL (localização do documento no sistema de arquivos) e retorna para o Gerenciador DSM-CC a composição NCM representando o documento.

O Gerenciador DSM-CC passa então ao Gerenciador de Bases Privadas uma solicitação para inserir a composição NCM na base privada que estiver aberta no momento. Ao receber um evento DSM-CC que transporta, no campo "nome do evento", o valor "*startNCL*", o Gerenciador DSM-CC solicita que a composição seja apresentada a partir da sua porta de entrada. Conforme definido no modelo NCM, a porta de entrada de uma composição é um mapeamento para um nó diretamente contido na composição e uma interface desse nó, que pode ser uma âncora do nó ou uma outra porta, se o nó interno for uma outra composição.

O sub-módulo Gerenciador de Bases Privadas permite também que, através do sub-módulo Gerenciador DSM-CC, um autor especifique (no provedor de conteúdo) modificações na estrutura das aplicações NCL (localizadas nos terminais de acesso). Para isso, o sub-módulo Gerenciador DSM-CC possui a capacidade de mapear, de forma trivial, os eventos definidos na Tabela 20 em chamadas aos métodos também oferecidos pela API de um *PrivateBaseManager*. Note que esses métodos não foram apresentados na Tabela 19, evitando repetir suas respectivas descrições.

| **Nome do Evento** | **Dados Específicos da Aplicação** | **Descrição** |
|---|---|---|
| addNode | compositeId, node | Insere nó em uma dada composição da base privada em uso. |
| removeNode | compositeId, nodeId | Retira nó de uma dada composição da base privada em uso. |
| addDescriptor | descriptor | Insere descritor na base privada em uso. |
| removeDescriptor | descriptorId | Remove descritor da base privada em uso. |
| addAnchor | nodeId, anchor | Insere âncora em um nó da base privada em uso. Caso a âncora exista, atualiza seus atributos. |
| removeAnchor | nodeId, anchorId | Remove e destrói âncora de um nó da base privada em uso. |
| addLink | compositeId, link | Insere elo em uma dada composição da base privada em uso. |
| removeLink | compositeId, linkId | Remove elo de uma dada composição da base privada em uso. |

Tabela 20: Eventos DSM-CC interpretados pelo sub-módulo Gerenciador DSM-CC.

É importante ressaltar que as modificações em uma aplicação declarativa, especificadas no provedor de conteúdo por um ambiente de autoria, são coerentemente atualizadas nos terminais de acesso, preservando todos os relacionamentos, incluindo aqueles que definem a estruturação lógica do documento NCL. Essas modificações podem ser realizadas em tempo de exibição.

Três observações devem ser feitas em relação aos métodos mapeados através da Tabela 20. A primeira observação é sobre o método *addLink*, que pode especificar uma âncora inexistente de um nó como ponto terminal do elo. Caso isto ocorra, uma âncora deve ser criada no nó, com seu conteúdo ainda indefinido. Esse conteúdo deverá ser preenchido, posteriormente, pelo comando *addAnchor*. Por exemplo, no caso em que a âncora represente um conjunto de unidades de

informação de um objeto de mídia, a âncora é criada com o conjunto vazio, a ser preenchido como indica a segunda observação.

A segunda observação é sobre o método *addAnchor*, que pode criar uma âncora totalmente nova ou substituir o conteúdo de uma âncora já criada. O conteúdo de uma âncora pode especificar o conteúdo de um atributo do nó, no caso da âncora ser um atributo. No caso da âncora especificar um conjunto de unidades de informação do nó, a âncora deve definir esse conjunto ou especificar um instante de tempo mais um deslocamento *d*. Essa última opção é usada para criar uma âncora em um nó que está sendo gerado e exibido ao vivo, cujo evento de apresentação ocorrerá no instante de tempo especificado no comando e durará *d* unidades de tempo.

Finalmente, a terceira observação é, novamente, sobre o método *addLink*. Permitir inserir elos em uma composição, significa, também, possibilitar que o provedor de conteúdo dispare, através da inserção de um elo com condição já satisfeita, a apresentação de um nó específico (relacionado por esse elo). Isso caracteriza uma alternativa para o processo de apresentação de um documento NCL. Note, que esse tipo de elo permite, na verdade, que o provedor de conteúdo execute qualquer ação de apresentação (i.e. *play*, *pause*, *stop* e *abort*) sobre os nós de uma base privada.

Para que o Núcleo seja capaz de apresentar as entidades NCM, presentes na base privada em uso, é necessário transformá-las em um conjunto, denominado contêiner, de entidades esperadas por ele. Novamente, o módulo Conversores é solicitado pelo sub-módulo Gerenciador de Bases Privadas. É interessante ressaltar que o módulo Conversores pode ser solicitado também para permitir que o ambiente de execução do Núcleo controle a apresentação de documentos especificados em outras linguagens como, por exemplo, documentos SMIL (Bulterman, 2004). Entretanto, para manter uma arquitetura leve, atendendo aos requisitos discutidos no Capítulo 1, foram definidos no *Maestro* apenas os conversores necessários à apresentação de documentos NCL (tradução da linguagem NCL para o modelo conceitual NCM e a tradução do modelo NCM para o ambiente de execução do Núcleo (Rodrigues, 2003)). Outros conversores (Rodrigues, 2003) são especificados como opcionais.

Um contêiner pode conter entidades como objetos de execução, elos e *layout*. O objeto de execução representa a instância de um nó a ser exibido,

definido no documento NCL, contendo todas as suas informações, inclusive as suas características de apresentação provenientes do descritor associado. Os *layouts* presentes em um contêiner consistem em janelas e regiões, definidas na autoria, para a apresentação dos nós. O módulo Gerenciador de *Layout*, apresentado na Figura 15, é responsável por mapear as janelas e regiões, definidas no documento NCL, nos componentes gráficos disponíveis no terminal de acesso. Através das janelas e regiões criadas, bem como de possíveis relacionamentos espaciais[8] definidos na autoria, o Gerenciador de *Layout* define as características espaciais dos exibidores e, conseqüentemente, dos objetos de execução.

As instanciações dos exibidores de mídia (exibidores dos objetos de mídia) são controladas pelo módulo Gerenciador de Exibidores, apresentado na Figura 15. Estratégias de adaptação ao contexto (por exemplo, opções de resolução segundo o perfil do usuário telespectador, entre outras (Rodrigues, 2003)) podem ser acionadas, através do módulo Adaptador, para determinar uma configuração que atenda restrições do autor ou do contexto em questão. Finalmente, o Núcleo inicializa o módulo Escalonador, responsável por orquestrar a apresentação dos objetos de execução, de acordo com as relações definidas na autoria.

É importante que o Núcleo e os Exibidores estejam fortemente integrados, pois são os exibidores que decodificam os conteúdos, recebidos via fluxo de transporte (através de DSM-CC ou fluxos elementares) ou via canal de retorno (por exemplo, por TCP/IP), e são capazes de perceber o ponto exato em que a exibição de um determinado objeto se encontra para informá-lo ao Núcleo. Além disso, as interações do usuário telespectador são realizadas sobre os exibidores e não com o Núcleo, cabendo a eles repassar as informações a respeito dessas interações. Assim, é definida em (Rodrigues, 2003) uma interface para troca de mensagens entre os módulos Núcleo e Exibidores, adotada na implementação da comunicação entre o núcleo do *Maestro* e todos os exibidores implementados.

Note, na Figura 15, que um navegador HTML está entre os exibidores definidos para o middleware *Maestro*. Isso significa que aplicações desenvolvidas na linguagem HTML podem ser apresentadas pelo *Maestro*. Além disso, a fim de permitir que código executável fosse integrado a programas exibidos no

[8] Relacionamentos espaciais podem mudar o posicionamento de objetos de execução.

middleware declarativo, um interpretador Lua (Ierusalimschy et al, 2003) foi incorporado para atribuir mais recursos à autoria de documentos NCL. Essa integração faz com que o middleware puramente declarativo seja estendido para funcionar como um middleware híbrido (declarativo+procedural). O uso de Lua oferece vantagens: além de ser um projeto de código aberto e desenvolvido em C, Lua combina programação procedural com poderosas construções para descrição de dados, baseadas em tabelas associativas e semântica extensível; Lua é tipada dinamicamente, interpretada a partir de *bytecodes*, e tem gerenciamento automático de memória com coleta de lixo. Segundo , Lua se destaca pelo alto desempenho e baixo consumo de recursos apresentados, quando comparada com outras linguagens interpretadas. Essas características fazem de Lua uma linguagem ideal para configuração, automação (*scripting*) e prototipagem rápida (Ierusalimschy et al, 2003). O próximo capítulo discorre sobre a implementação da versão corrente do middleware *Maestro*.

# 5
# Descrição da Implementação

A implementação do middleware declarativo *Maestro* teve início com a adaptação do Formatador HyperProp (Rodrigues, 2003), originalmente implementado na linguagem Java (Rodrigues, 2003), ao contexto da TV Digital. Para atender aos objetivos deste trabalho, discutidos no Capítulo 1, o formatador adaptado foi implementado na linguagem C++, utilizando uma abordagem orientada a objetos e visando plataformas que utilizam o sistema operacional Linux.

Os módulos Filtro de Seções, Sintonizador e DSM-CC, foram modelados de acordo com as discussões do capítulo anterior. Porém, a implementação desses módulos foi realizada pelo laboratório Lavid da Universidade Federal da Paraíba, dentro da proposta do middleware procedural FlexTV , por ela desenvolvido. A integração desses módulos ao middleware declarativo *Maestro* faz parte do processo de integração definido no projeto SBTVD. Os sub-módulos Gerenciador de Bases Privadas e Gerenciador DSM-CC foram modelados de acordo com as discussões realizadas no capítulo anterior, mas suas implementações foram deixadas para trabalhos futuros.

Como prova de conceito o middleware declarativo proposto foi integrado ao perfil simples de terminal de acesso definido pelo SBTVD, denominado Kalaheo, com a seguinte configuração: processador Celeron 700 MHz e 128 MB de memória RAM. O middleware foi também integrado a um terminal de acesso (Geode) de configuração inferior: processador AMD 233 MHz e 64 MB de RAM. A implementação do middleware com todas as suas funcionalidades foi testada em um computador pessoal, simulando um terminal de acesso de grande poder computacional: processador Pentium IV de 3.1 GHz e 1 GB de memória RAM.

As seções seguintes discorrem sobre as bibliotecas utilizadas pelo *Maestro* e a implementação do Núcleo de sua arquitetura modular, bem como dos seus exibidores. A versão atual dos módulos Núcleo e Exibidores do middleware *Maestro* conta com aproximadamente 280 classes. Maiores detalhes da

implementação podem ser encontradas em: http://www.telemidia.puc-rio.br/~marcio.

## 5.1. Bibliotecas Utilizadas

Além de DirectFB, discutida no Capítulo 2, foram utilizadas, na implementação do middleware *Maestro*, as seguintes bibliotecas: liburi, libxerces-c, liblua, libxine, libfusionsound, libpng, libjpeg e libmbrowser. Todas as bibliotecas utilizadas são livres e de código aberto.

A biblioteca liburi[9] oferece uma abstração para resolução de endereços dos documentos NCL a serem entregues ao Núcleo. Desenvolvida na linguagem C, a biblioteca foi projetada com o objetivo de obter um tempo mínimo de resposta e demandar pouco espaço no disco rígido. A função principal da biblioteca consiste em transformar uma URI (*Universal Resource Identifier*) específica em uma estrutura em C e vice-versa. A estrutura em C possui um campo para cada componente de uma URI: *scheme*, *path*, *query*, *params* e *host*.

Para realizar o *parser* de documentos NCL é utilizada a biblioteca libxerces-c[10], uma biblioteca portável e implementada na linguagem C++. Essa biblioteca foi desenvolvida com o objetivo de oferecer funcionalidades para a interpretação, validação, manipulação e criação de documentos XML. O *parser* da biblioteca libxerces-c foi desenvolvido com foco no desempenho, modularidade e escalabilidade.

Com o objetivo de interpretar códigos procedurais escritos na linguagem Lua, foi incorporada a biblioteca liblua[11]. Essa biblioteca é implementada como uma pequena biblioteca de funções C, escritas em ANSI C, que compila sem modificações na maioria das plataformas conhecidas. Os objetivos da implementação são simplicidade, eficiência, portabilidade e baixo impacto de inclusão em aplicações.

---

[9] http://www.senga.org

[10] http://xml.apache.org/xerces-c/

[11] http://www.lua.org

Normalmente, as decodificadoras MPEG-2 dos terminais de acesso para TV Digital não possuem capacidade para decodificar o programa principal e um objeto de vídeo simultaneamente. Dessa forma, para permitir a exibição de objetos de vídeo em paralelo com a exibição do programa principal decodificado por hardware, é utilizada a biblioteca libxine[12]. Essa biblioteca permite também que vídeos (inclusive do programa principal) sejam decodificados em plataformas que não possuem hardware de decodificação MPEG. A biblioteca libxine é implementada na linguagem C e possui capacidade para decodificar objetos de mídia conformes aos padrões MPEG-1 (ISO, 1993) e MPEG-2 (ISO, 2000a), entre outros (Xine, 2006).

A biblioteca libfusionsound[13] é utilizada para decodificar objetos isolados de áudio (i.e. que não foram multiplexados com objetos de vídeo). A biblioteca é implementada na linguagem C e possui capacidade para decodificar objetos de áudio concomitantemente com a apresentação de outro objetos, incluindo o vídeo principal.

As bibliotecas libpng e libjpeg são utilizadas para decodificar imagens estáticas nos formatos PNG (*Portable Network Graphics*) e JPEG (*Joint Photographic Experts Group*), respectivamente. Ambas foram implementadas na linguagem C e, geralmente, são oferecidas nos pacotes de instalação dos sistemas Linux.

Finalmente, a biblioteca libmbrowser foi desenvolvida neste trabalho com o objetivo de integrar um exibidor de objetos HTML ao middleware *Maestro*. Essa biblioteca é baseada no navegador links[14], que foi adaptado ao contexto da TV Digital, como será discutido na Seção 5.3.4. O navegador links foi implementado na linguagem C e é capaz de interpretar páginas HTML 4.0, com suporte a CSS e JavaScript.

---

[12] http://www.xinehq.de

[13] http://www.directfb.org

[14] http://links.twibright.com

## 5.2. Núcleo

A estrutura central da implementação atual do middleware *Maestro* é representada pelo diagrama de classes apresentado na Figura 16. A entidade principal do middleware implementado é a classe `Formatter`, que corresponde ao módulo Núcleo da arquitetura do middleware *Maestro*, definida no capítulo anterior. Note, através da Figura 16, que um objeto dessa classe possui, obrigatoriamente, um escalonador (classe `FormatterScheduler`) e um adaptador (sub-sistema `Adapters`), que correspondem, respectivamente, aos sub-módulos Escalonador e Adaptador. Além disso, um objeto `Formatter` também mantém um contêiner (classe `Container`), discutido na Seção 4.5 e, opcionalmente, um layout (classe `FormatterLayout`), que é mantido pelo sub-módulo Gerenciador de Layout. Finalmente, para cada modelo (ou linguagem) de documentos que o objeto `Formatter` aceita como entrada (descrevendo os programas a serem apresentados), deve haver uma implementação de conversor (sub-sistema `Converters`) para o modelo de execução. Dessa forma, conversores de documentos NCL para o modelo NCM e do modelo NCM para o modelo de execução do Núcleo foram implementados.

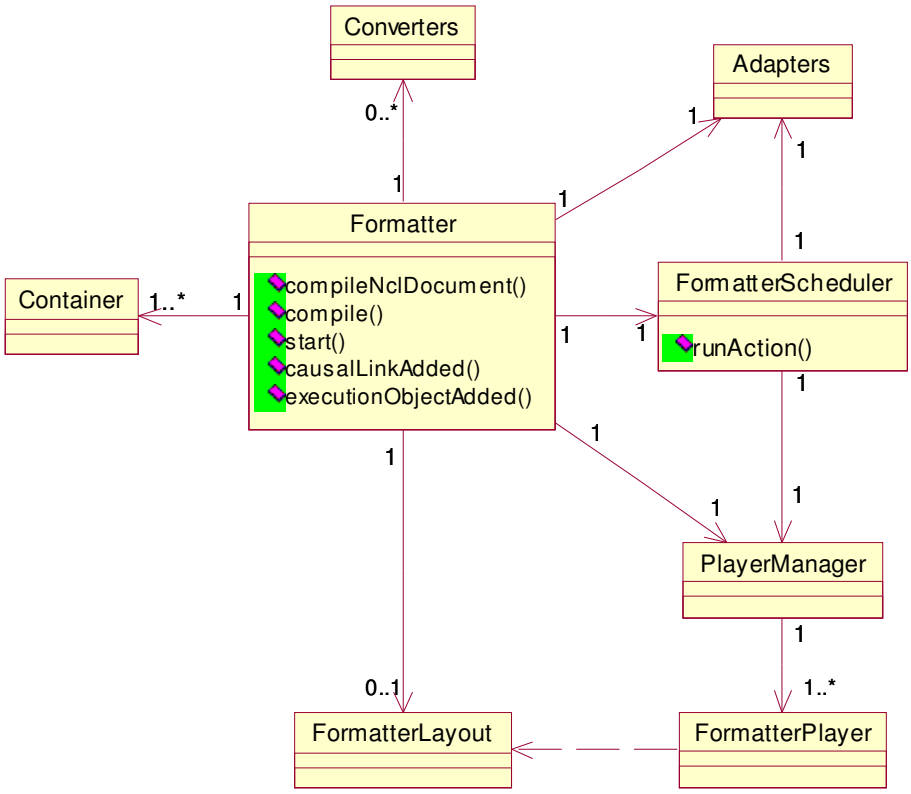


Figura 16: Diagrama de Classes central do Núcleo do *Maestro* com principais métodos.

A primeira etapa na apresentação de um documento NCL é a chegada de uma solicitação para o início da sua exibição. Essa solicitação, que atualmente é proveniente de uma interface externa ao middleware *Maestro*, consiste em chamadas a três métodos do objeto `Formatter`: *compileNclDocument()*, na qual a descrição do documento (ou parte dele) é passada como parâmetro; *compile()*, na qual um contêiner é passado como parâmetro; e *start()*.

O método *compileNclDocument()* é responsável por retornar um contêiner, resultado da tradução da especificação NCL para o modelo de execução do Núcleo. Essa tradução é realizada pelo sub-sistema Conversores. O contêiner obtido é passado como parâmetro na chamada ao método *compile()*.

O método *compile()* é responsável por incluir no container do objeto `Formatter`, todas as entidades presentes no contêiner recebido (possivelmente: elos, objetos de execução e layout). O objeto `Formatter` é um observador do seu contêiner. Assim, cada elo causal (elos que possuem uma condição específica que, quando satisfeita, implica no disparo de uma ação) (Rodrigues, 2003) adicionado no contêiner do objeto `Formatter`, gera uma notificação. Essa notificação

consiste em uma chamada a um método do objeto `Formatter`, denominado *causalLinkAdded()*. Esse método é responsável por registrar o objeto `FormatterScheduler` como observador desses elos. Além disso, cada objeto de execução inserido no container do formatador gera uma notificação, que consiste em uma chamada a um método do objeto `Formatter`, denominado *executionObjectAdded()*, passando como parâmetro o objeto de execução adicionado. Nesse método, são avaliadas, caso existam, as alternativas de objetos de execução que podem ser resolvidas estaticamente (antes de iniciar a apresentação). As alternativas de objetos de execução são associadas a regras que devem ser avaliadas. Dessa forma, para avaliação das alternativas, o serviço de um objeto do sub-sistema `Adapters` é requisitado. Como as regras podem depender de parâmetros definidos na plataforma do terminal de acesso (capacidade de processamento, recursos disponíveis etc.), assim como preferências do usuário telespectador, o adaptador consulta o contexto de apresentação através de um componente denominado `ContextInfoProxy`. Na implementação corrente, esse *proxy* de informações contextuais é uma implementação simples de gerente de contexto que mantém informações da plataforma, carregadas a partir de arquivos textos locais.

Ainda no método *executionObjectAdded()*, se houver um descritor associado ao objeto de execução (recebido como parâmetro), o objeto `FormatterLayout` pode ser solicitado para converter as informações obtidas das janelas e regiões do layout do documento NCL para superfícies (interface *IDirectFBSurface* que faz parte de uma interface *IDirectFBWindow*) e sub-superfícies (estrutura recebida através da referência *IDirectFBSurface->GetSubSurface()*) oferecidas pela biblioteca DirectFB. Além disso, no método *executionObjectAdded()* é solicitado ao objeto `PlayerManager`, apresentado no diagrama de classes da Figura 16, que crie um exibidor (classe `FormatterPlayer`, na Figura 16) apropriado para o objeto de execução (recebido como parâmetro). O objeto `PlayerManager` corresponde ao sub-módulo Gerenciador de Exibidores.

Finalmente, o método *start()* (do objeto `Formatter`) é responsável por disparar o escalonador (através de uma chamada ao método *start()* do objeto `FormatterScheduler`), passando como parâmetro um evento de apresentação para iniciar a exibição dos objetos de execução presentes no contêiner recém compilado no objeto `Formatter`.

Durante a apresentação do documento, o objeto `FormatterScheduler` permanece como um observador dos elos causais, sendo notificado sempre que uma condição for satisfeita. Essa notificação consiste em uma chamada ao método *runAction()* (do objeto `FormatterScheduler`), passando como parâmetro a ação especificada no elo causal satisfeito. Esse método é responsável por coordenar o disparo da ação (recebida por parâmetro), bem como verificar a necessidade de um objeto de execução ter o evento de apresentação de seu conteúdo preparado, ou mesmo ter sua ocorrência iniciada.

A próxima seção trata em mais detalhes a instanciação dos exibidores de mídia, durante a apresentação dos documentos NCL, bem como as particularidades na implementação realizada de exibidores para o middleware *Maestro*.

## 5.3. Exibidores

No middleware *Maestro* foram implementados exibidores para tratar os seguintes tipos de conteúdo: HTML, Lua, imagens estáticas (JPEG, GIF e PNG), áudio (WAVE, MPEG-1 e MPEG-2) e vídeo (MPEG-1 e MPEG-2). A Figura 17 ilustra um diagrama de classes resumido para a implementação dos exibidores.

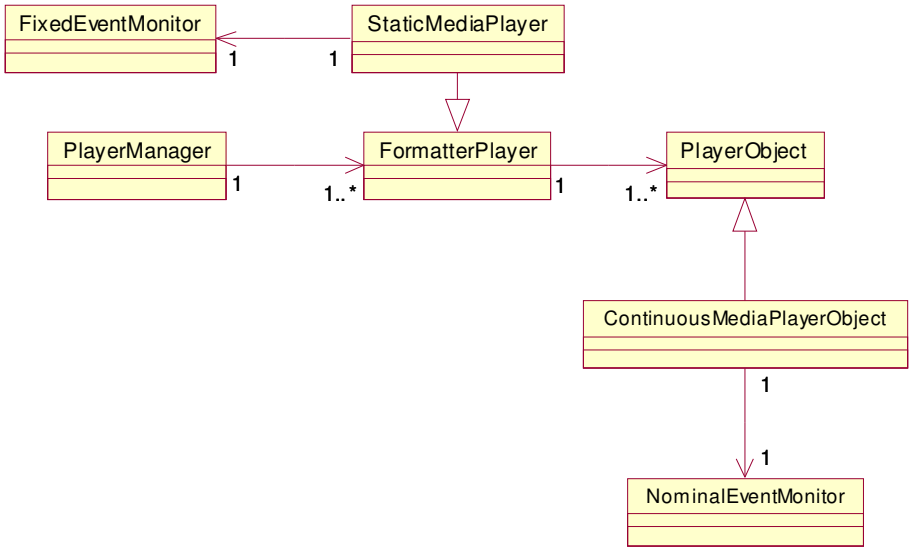


Figura 17: Diagrama de Classes para a implementação dos Exibidores.

Todo exibidor implementado deve especializar, direta ou indiretamente, os atributos e métodos da classe `FormatterPlayer`. Esses métodos permitem ao

Núcleo requisitar a execução das ações sobre eventos controlados pelo exibidor (Rodrigues, 2003). O Núcleo do middleware *Maestro* prevê que um mesmo exibidor possa ser utilizado para controlar a apresentação de mais de um objeto, esses objetos de exibição são modelados pela classe `PlayerObject`. Os exibidores de mídia contínua devem especializar a classe `ContinuousMediaPlayerObject`, que possui uma instância da classe `NominalEventMonitor`. Essa classe é responsável por monitorar temporalmente trechos do conteúdo apresentado pelo objeto que especializa a classe `ContinuousMediaPlayerObject`. Já os exibidores de mídia estática devem especializar a classe `StaticMediaPlayer`, que possui uma instância da classe `FixedEventMonitor`. Essa classe é responsável por monitorar eventos específicos na apresentação do objeto que especializa a classe `StaticMediaPlayer`.

A forma com que os exibidores foram implementados e como utilizam o diagrama de classes ilustrado na Figura 17 é discutida nas seções a seguir.

### 5.3.1. Áudio e Vídeo

O exibidor de áudio e vídeo foi implementado através das classes `AVPlayerAdapter`, `AVPlayerObject` e `AVPlayer`.

A classe `AVPlayer` foi implementada de forma a reconhecer um dispositivo de decodificação MPEG-2 (i.e. decodificação MPEG-2 por hardware) e controlar a exibição de fluxos de áudio e vídeo por esse dispositivo. Além disso, a classe `AVPlayer` foi implementada de tal forma a tratar também a decodificação e exibição de fluxos MPEG por software. Isso significa que o exibidor de áudio e vídeo permite que fluxos (inclusive do programa principal) sejam apresentados em plataformas que não possuem hardware de decodificação MPEG, ou que vídeos e áudios possam ser exibidos paralelamente ao programa principal decodificado por hardware. Evidentemente, a resolução dos vídeos decodificados por software irá depender da capacidade de memória e processamento do terminal de acesso.

Instâncias `AVPlayer`, que processam fluxos audiovisuais decodificados por software, utilizam a interface *IDirectFBVideoProvider* (da biblioteca DirectFB) para renderizar o conteúdo visual do fluxo na superfície definida através do método *setSurface()* (superfície definida na camada gráfica do modelo de

apresentação). A interface *IDirectFBVideoProvider* foi integrada à biblioteca libxine para decodificar os objetos MPEG. Para iniciar a exibição do fluxo decodificado, um método denominado *playVideo()* é utilizado.

Instâncias `AVPlayer` que processam fluxos audiovisuais por hardware possuem um funcionamento mais complexo. Para possibilitar a exibição do fluxo de áudio e vídeo principal decodificado por hardware simultaneamente aos outros exibidores, foram utilizadas as funcionalidades de uma API do sistema operacional Linux, conhecida como Video4Linux (Cox, 2000).

No contexto da API Video4Linux, todo o conteúdo gerado pela decodificação por hardware é enviado para um dispositivo denominado "dispositivo de vídeo" (*video*) (Cox, 2000). Esse dispositivo é utilizado, através de chamadas definidos pela API Video4Linux, para o controle de dispositivos de entrada e saída `ioctl` (Cox, 2000). Nesse caso específico a chamada para iniciar o uso do dispositivo é `open("/dev/video", O_RDONLY)`, que retorna um número inteiro que funciona como um descritor. No entanto, para exibir as informações no dispositivo de vídeo, é necessário abrir para utilização um outro dispositivo denominado "saída do dispositivo de vídeo" (*vout*), através da chamada `open("/dev/vout", O_RDONLY)`, que também retorna um descritor representado por um número inteiro.

Uma vez aberto o dispositivo *vout*, é necessário definir uma área na tela para que as informações sejam exibidas. Essa área é definida através da especificação de uma cor chave. A área utilizada na implementação é definida por uma interface *IDirectFBSurface*, possuindo a cor magenta, uma vez que essa é a cor chave padrão definida na API Video4Linux. Assim, essa superfície (que faz parte de uma *IDirectFBWindow*) consiste na camada de vídeo do modelo de apresentação.

Para conseguir sobrepor algum conteúdo sobre a camada de vídeo, é necessário, segundo a API Video4Linux (Cox, 2000), definir algumas informações denominadas de informações de *overlay*. Isso é realizado através da seguinte chamada: `ioctl(fd_vout, IOCTL_SET_OVERLAY_INFO, &ovr_info)`. Nessa chamada as informações de *overlay* são especificadas através da estrutura `ovr_info`. Resumindo, através das informações de *overlay* (cor chave da camada de vídeo, entre outras (Cox, 2000)) é definida a camada gráfica do modelo de apresentação. Porém, quando deseja-se eliminar (terminar, esconder, etc.) o conteúdo que sobrepõe a camada de vídeo, é necessário refazer as definições

dessa camada, uma vez que foi perdida a cor chave outrora presente na área utilizada pelo exibidor desse conteúdo específico. Para isso, a API Video4Linux permite definir informações de *alpha_blend* (área, forma de transição entre o conteúdo da camada gráfica e da camada de vídeo (Cox, 2000)), através da chamada: `ioctl(fd_vout, GEODE_IOCTL_SET_ALPHA_BLEND_INFO, &alp_info)`. As informações alpha_blend foram utilizadas também, na implementação da classe `AVPlayer`, para redimensionamento da área de apresentação do vídeo exibido na camada de vídeo.

O adaptador de áudio e vídeo (classe `AVPlayerAdapter`) foi implementado com o objetivo de compatibilizar o exibidor de áudio e vídeo implementado (classe `AVPlayer`), com a API de troca de mensagens entre os módulos Exibidores e Núcleo (Rodrigues, 2003). Entre as principais funções dos adaptadores de áudio e vídeo estão: sinalizar ao Núcleo eventos de apresentação (início da exibição da região de uma âncora, fim da exibição da região da âncora etc.) e executar ações sobre a apresentação como, por exemplo, aumentar volume do áudio, suspender a apresentação, reiniciar, acelerar etc., comandadas pelo Núcleo.

O adaptador de áudio e vídeo é uma especialização da classe `FormatterPlayer`, podendo controlar um ou mais objetos de exibição de áudio e vídeo (instâncias da classe `AVPlayerObject`, que especializa a classe `ContinuousMediaPlayerObject`).

Conforme discutido na Seção 5.2, durante a apresentação de um documento NCL, quando um exibidor apropriado deve ser selecionado para exibir um objeto de execução, o objeto `PlayerManager` é consultado. O passo seguinte é solicitar ao exibidor (instância de uma classe que herde de `FormatterPlayer`) a preparação da apresentação do objeto, através de uma chamada ao método *prepare()*. Esse método é responsável, entre outras coisas (Rodrigues, 2003), pela criação de uma instância de objeto de exibição de áudio e vídeo (através do método *createPlayerObject()* do adaptador).

O objeto de exibição de áudio e vídeo (`AVPlayerObject`) cria, em seu método construtor, um objeto `AVPlayer`, passando como parâmetro uma referência (obtida através do objeto de execução) para o fluxo de áudio e vídeo a ser exibido. Além disso, o objeto de exibição de áudio e vídeo pode definir no objeto `AVPlayer` (através do método *setSurface()* implementado) uma região (i.e. uma superfície) especificada pelo descritor obtido também através do objeto de

execução. Uma vez preparada a exibição, as ações de apresentação de iniciar (*start*), pausar (*pause*), retomar (*resume*) e encerrar (*stop* ou *abort*) a apresentação são mapeados do adaptador de áudio e vídeo para o objeto `AVPlayer` de forma trivial.

O objeto de exibição de áudio e vídeo monitora temporalmente a apresentação do conteúdo (através da classe `NominalEventMonitor`). Para isso, uma pthread (Nichols et al, 1996) é criada para observar e sinalizar ocorrências temporais passadas inicialmente como parâmetro para o monitor. Fazendo uso de chamadas ao método *getMediaTime()* do objeto de exibição de áudio e vídeo, esse monitor observa se um determinado trecho teve sua exibição iniciada ou terminada, para sinalizar tal fato ao Núcleo. A classe `AVPlayerObject` implementa essa chamada simplesmente consultando a instância de `AVPlayer` através de um método com a mesma assinatura.

### 5.3.2. Imagens Estáticas

O exibidor de imagens estáticas foi implementado através das classes `ImagePlayer` e `ImagePlayerObject`. Para utilizar esse exibidor, as imagens estáticas podem ser enviadas no mesmo carrossel que o documento NCL é transmitido, ou mesmo serem obtidas via canal de retorno. Nos testes da implementação, as imagens foram dispostas diretamente no sistema de arquivos.

A classe `ImagePlayer` foi implementada como uma especialização da classe `StaticMediaPlayer`, podendo instanciar e controlar um ou mais objetos de exibição de imagens estáticas (instâncias da classe `ImagePlayerObject`, que especializa a classe `PlayerObject`).

Para iniciar a exibição de uma imagem estática, é utilizado o método *start()* de um objeto `ImagePlayer`, que recebe como parâmetro um objeto de execução. Nesse método é criada uma instância que implementa a interface *IDirectFBImageProvider*, através da biblioteca DirectFB. Essa instância é utilizada para renderizar (na camada gráfica do modelo de apresentação) a imagem estática, cuja referência é obtida através do objeto de execução, na região (sub-superfície) especificada pelo descritor obtido, também, através do objeto de execução passado como parâmetro. No processo de renderização, a interface

*IDirectFBImageProvider* utiliza as bibliotecas libpng e libjpeg para decodificar a imagem estática especificada.

Para finalizar a exibição de uma imagem estática, é utilizado o método *stop()* de um objeto `ImagePlayer`, responsável por liberar os recursos utilizados na exibição. Existe ainda a possibilidade do autor, na especificação de um documento NCL, definir um tempo explícito de exibição da imagem estática. Nesse caso, é função do objeto `FixedEventMonitor` controlar esse tempo e sinalizar para que a exibição seja encerrada quando decorrida duração.

### 5.3.3. Lua

O exibidor Lua foi implementado através das classes `LuaPlayer` e `LuaPlayerObject`. Para utilizar o exibidor Lua, um arquivo, contendo código procedural especificado nessa linguagem, pode ser enviado no mesmo carrossel que o documento NCL é transmitido. No entanto, nos testes da implementação, arquivos contendo código Lua foram dispostos diretamente no sistema de arquivos. Além disso, na especificação do documento NCL um nó de mídia do tipo “Lua”, deve fazer referência a esse tipo de arquivo específico. Esse nó pode referenciar funções Lua através de atributos.

A classe `LuaPlayer` foi implementada, inicialmente, como uma especialização da classe `StaticMediaPlayer`, podendo instanciar e controlar um ou mais objetos de exibição Lua (instâncias da classe `LuaPlayerObject`, que especializa a classe `PlayerObject`). Ao ser instanciado, um objeto de exibição Lua solicita a iniciação do ambiente Lua (liblua), caso o mesmo não esteja iniciado.

Para iniciar a interpretação do código procedural Lua, é utilizado o método *start()* de um objeto de exibição Lua. Existe a possibilidade de exibir componentes gráficos relacionados com a interpretação do código Lua. Assim, o código procedural Lua pode utilizar uma interface *IDirectFBFont* (da biblioteca DirectFB). A exibição é realizada em uma região (sub-superfície), na camada gráfica do modelo de apresentação, também especificada pelo código procedural Lua. As chamadas às funções Lua existentes no código procedural interpretado podem ser realizadas através de elos. Esses elos podem conter atributos a serem passados como parâmetros para as funções Lua.

Para finalizar o ambiente Lua, bem como a possível exibição do resultado da interpretação de um código Lua específico, é realizada uma chamada ao método *stop()* de um objeto `LuaPlayer`. Esse método tem o objetivo de fazer com que o exibidor Lua demande o mínimo de recursos possíveis, finalizando o ambiente Lua e liberando os recursos utilizados na possível exibição do resultado. Existe ainda a possibilidade do autor, na especificação de um documento NCL, definir um tempo explícito de exibição do resultado. Nesse caso, é função do objeto `FixedEventMonitor` controlar esse tempo e sinalizar para que a exibição seja encerrada quando decorrida duração.

### 5.3.4. HTML

O navegador links[15] foi a base para implementação do navegador HTML do middleware *Maestro*. Entretanto, o código do navegador links precisou ser alterado, porque sua implementação não permite que uma aplicação usuária especifique a região exata na tela onde a janela do navegador deve ser aberta. Além disso, o navegador links não foi implementado como uma aplicação do tipo multi-aplicação (i.e. aplicações que permitem que a interface gráfica seja compartilhada por outras aplicações). Para resolver essas limitações, foram realizadas as seguintes modificações: cada instância do navegador passou a consistir em uma nova *pthread*; e a região que o navegador utiliza passou a poder ser passada como parâmetro, utilizando a classe `FormatterLayout`.

O exibidor HTML foi implementado através das classes `TextPlayer` e `TextPlayerObject`. Para utilizar o exibidor HTML, um nó do documento NCL, deve ser do tipo "*text*".

De forma análoga aos exibidores de mídia estática, discutidos nas seções anteriores, a classe `TextPlayer` foi implementada como uma especialização da classe `StaticMediaPlayer`, podendo instanciar e controlar um ou mais objetos de exibição HTML (instâncias da classe `HTMLPlayerObject`, que especializa a classe `PlayerObject`).

---

[15] http://links.twibright.com/

Para iniciar a exibição de uma página HTML, é utilizado o método *start()* de um objeto `TextPlayer`, que recebe como parâmetro um objeto de execução. Nesse método é criada uma instância do navegador, através da biblioteca libmbrowser, passando como parâmetro a URL, obtida através do objeto de execução, bem como uma região (sub-superfície) onde o navegador será exibido, especificada pelo descritor, também obtido através do objeto de execução.

O método *stop()* de um objeto `TextPlayer` pode ser utilizado para finalizar a exibição de uma página HTML. Esse método é responsável por liberar os recursos utilizados na exibição da página. Caso o autor, na especificação de um documento NCL específico, defina um tempo explícito de exibição da página HTML, é função do objeto `FixedEventMonitor` controlar a exibição da página, como discutido nas seções anteriores.

### 5.3.5. Eventos do Usuário Telespectador

Conforme discutido na Seção 5.2, um objeto `FormatterLayout`, presente no diagrama de classes da Figura 18, pode ser solicitado para converter as informações de posicionamento espacial, obtidas dos próprios objetos de execução, para superfícies (controladas pela classe `FormatterWindow`) e sub-superfícies (controladas pela classe `FormatterRegion`) oferecidas pela biblioteca DirectFB. Ao ser instanciada, uma janela `FormatterWindow` solicita a um gerenciador de eventos (classe `EventManager`, apresentada no diagrama de classes da Figura 18) uma referência para um *buffer* responsável por manter eventos gerados pelo usuário telespectador. Esse gerenciador de eventos é obtido através do método *getInstance()*, uma vez que a classe `EventManager` foi implementada como *singleton*. Para isso, um objeto `EventManager` mantém um *buffer* (*IDirectFBEventBuffer*, introduzido no Capítulo 3), responsável por receber eventos de todos os dispositivos de entrada (teclado, mouse, controle remoto etc.) presentes na plataforma.

Quando uma exibição é iniciada, pelo método *start()* de uma das especializações de `FormatterPlayer`, discutidas nas seções anteriores, um controlador de eventos do usuário telespectador (classe `KeyHanler`, apresentada no diagrama de classes da Figura 18) é instanciado. Esse objeto especializa a classe `KeyListener` (segundo o diagrama de classes da Figura 18), e cadastra-se

no gerenciador de eventos como *listener* de eventos do usuário telespectador, através do método *addKeyEventListener()* do objeto `EventManager`.

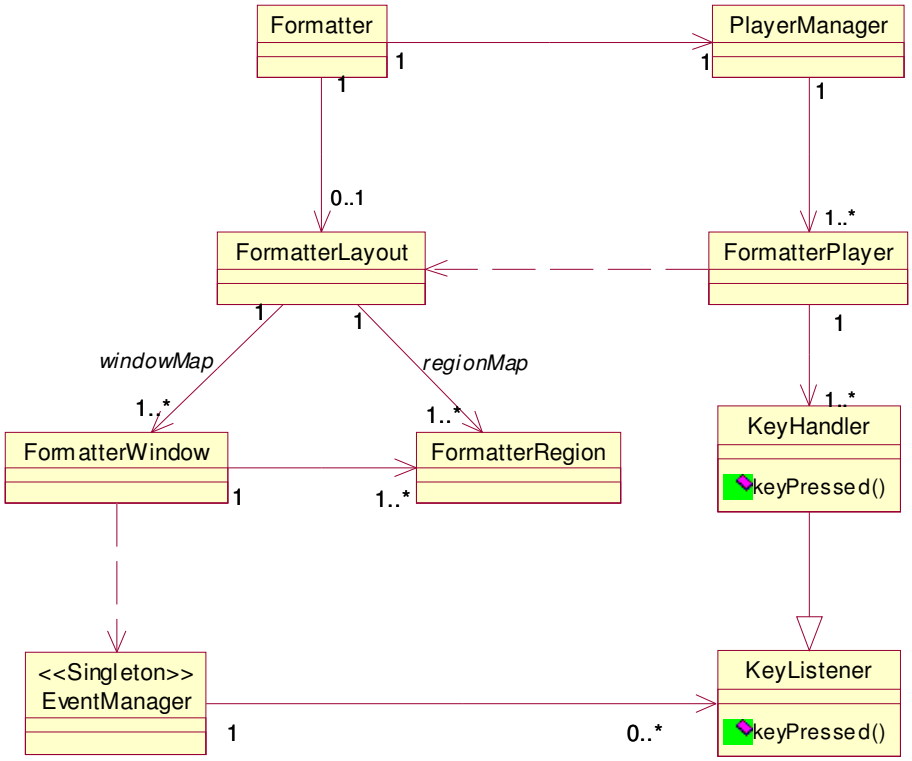


Figura 18: Diagrama de Classes do Gerenciador de Layout.

Quando uma janela `FormatterWindow` é exibida, uma *pthread* é criada para esperar eventos do usuário telespectador, que possivelmente serão gerados pelos dispositivos de entrada da plataforma, através da janela que possuir o foco (qualquer evento gerado em uma região é reportado à janela, uma vez que cada região é uma sub-superfície da superfície da janela). Essa *pthread* existe apenas enquanto a janela estiver em exibição. Ao ser gerado um evento, em qualquer janela que tiver o foco, todos os *listeners* de eventos do usuário telespectador são notificados pelo gerenciador de eventos, passando esse evento como parâmetro através de uma chamada ao método *keyPressed(),* implementado por esses *listeners* (objetos *KeyHandler*). Esse método verifica se o objeto de execução, passado como parâmetro ao instanciar um `KeyHandler`, possui alguma âncora relacionada ao evento recebido. Isso é feito para disparar um evento de seleção específico, que será tratado pelo objeto `FormatterScheduler`, conforme discussões realizadas na Seção 5.2.

Tudo isso significa que, independente de qual janela tenha o foco, todas as entidades em exibição serão notificadas do evento de usuário telespectador gerado, garantindo que as ações relacionadas a ocorrência do evento, naquele instante, sejam realizadas de acordo com o especificado na autoria.

# 6
# Conclusões e Trabalhos Futuros

Obter baixo custo nos terminais de acesso é fator crucial para o sucesso da TV digital aberta, principalmente nos países em desenvolvimento. Para fazer com que o baixo custo comprometa o mínimo possível dos recursos dos terminais de acesso, é importante que seja feita uma economia nos custos relativos à propriedade intelectual e a royalties associados ao software presente nesses dispositivos como, por exemplo, o middleware. Os principais middlewares existentes, além de apresentarem esses custos associados, possuem alto requisito de processamento, por estarem atrelados a tecnologias como máquinas virtuais Java. Para resolver essas e outras questões relacionadas ao contexto de TV digital, a implementação de um middleware declarativo, independente do uso de tecnologia Java, fez-se necessária.

Os principais middlewares declarativos existentes privilegiam a interatividade em detrimento da sincronização. Foi discutido que o sincronismo de mídias deve ser o foco da linguagem declarativa a ser utilizada pelo middleware, tratando a interatividade como um caso particular do sincronismo.

Este trabalho apresenta uma análise detalhada das principais questões envolvidas no planejamento, desenvolvimento e implementação de middlewares declarativos para TV digital interativa, podendo servir como uma base para projetos com foco nessa área.

Outra contribuição deste trabalho foi a implementação do middleware *Maestro*, um middleware declarativo que possui foco no sincronismo de mídias, através da linguagem NCL, e faz parte do modelo de referência proposto ao Sistema Brasileiro de TV Digital (SBTVD). Além de demandar baixo poder de processamento das máquinas, todas as tecnologias utilizadas nessa implementação estão livres de encargos adicionais, fazendo com que a implementação do middleware *Maestro* seja uma alternativa interessante, também, para perfis simples de terminais de acesso.

Em terminais com maior poder de processamento, o middleware *Maestro* pode, inclusive, trabalhar em conjunto com middlewares procedurais, existindo também a possibilidade do mesmo ser adicionado em uma plataforma que não suporte aplicações NCL de modo nativo (sem uso do *Maestro*).

Como prova de conceito, o middleware foi integrado no perfil simples de terminal de acesso definido pelo SBTVD, denominado Kalaheo, com a seguinte configuração: processador Celeron 700 MHz e 128 MB de memória RAM. O middleware foi também integrado a um terminal de acesso, denominado Geode, de configuração inferior: processador AMD 233 MHz e 64 MB de RAM. A potencialidade de todos os seus recursos foram verificadas através de sua implementação em um computador pessoal, simulando um terminal de acesso de grande poder computacional (processador Pentium IV de 3.1 GHz e 1 GB de memória RAM). Diversas aplicações NCL foram testadas no middleware *Maestro* em todas as versões implementadas.

Como decorrência da implementação do *Maestro*, outra contribuição deste trabalho foi a implementação de um Formatador NCL, e seus exibidores, na linguagem C++, uma vez que o mesmo é um subconjunto da arquitetura modular do middleware.

A pesquisa realizada nesta dissertação também contribuiu na definição de um mecanismo para possibilitar que modificações no documento, especificadas pelo autor em tempo de exibição, sejam refletidas na apresentação e possam posteriormente ser armazenadas, preservando todos os relacionamentos, incluindo aqueles que definem a estruturação lógica de um documento (Moreno et al, 2005c).

Como trabalho futuro, pretende-se implementar, sobre o middleware *Maestro*, outros exibidores, como, por exemplo, exibidores com capacidade de apresentar arquivos Flash (programa gráfico vetorial utilizado para criar animações interativas, desenvolvido e comercializado pela Macromedia[16]), através da integração da biblioteca libflash à biblioteca DirectFB. O objetivo, nesse caso, é explorar os recursos das animações Flash em conjunto com os recursos da linguagem NCL.

Outro trabalho futuro é a implementação de um encapsulador de pacotes IP sobre Fluxo de Transporte MPEG-2 (Moreno, 2005a), bem como a implementação, sobre o middleware *Maestro*, de um módulo desencapsulador de pacotes IP sobre MPEG-2 (Moreno, 2005a). A idéia central é possibilitar o uso do sistema de TV digital como instrumento complementar na inclusão digital. Como conseqüência, a partir de canais de retorno de baixo custo como, por exemplo, mensagens SMS (*Short Message Service*) por meio de um dispositivo móvel simples, usuários telespectadores poderão solicitar URLs e receber os respectivos conteúdos através da rede de difusão de TV digital (Moreno, 2005a).

Um quarto trabalho importante é a implementação dos módulos Gerenciador de Bases Privadas e Gerenciador DSM-CC, discutidos no Capítulo 4. A implementação de um módulo responsável por tratar fluxos de mídia contínua sobre o protocolo RTP (*Real-Time Transport Protocol*) consiste em outro trabalho futuro. Esse protocolo seria o responsável por realizar o transporte com sincronismo entre diferentes mídias, como áudio, vídeo e dados associados, através do canal de retorno. Além disso, é interessante definir como trabalho futuro o aperfeiçoamento dos testes realizados com o objetivo de medir questões de consumo de recursos e avaliações de desempenho do middleware Maestro.

Por fim, espera-se que, com a manipulação dos componentes desenvolvidos no middleware *Maestro*, sejam desenvolvidas novas implementações objetivando o seu funcionamento em conjunto com implementações dos principais middlewares procedurais existentes, bem como uma integração com o middleware procedural FlexTV.

---

[16] http://www.macromedia.com

# 7
# Referências


ARIB. **ARIB STD-B24, Version 3.2, Volume 3: Data Coding and Transmission Specification for Digital Broadcasting**, ARIB Standard, 2002.

ARIB. **ARIB STD-B23, Application Execution Engine Platform for Digital Broadcasting**. ARIB Standard, 2004.

ATSC. **A/93: Synchronized/Asynchronous Trigger Standard**, Abril de 2002. Disponível em *www.atsc.org/standards/a_93.pdf*. Acesso em 02/10/2004.

ATSC. **DTV Application Software Environment Level 1 (DASE-1) PART 2: Declarative Applications and Environment**, 2003.

ATSC. **Advanced Common Application Platform (ACAP)**, A/101. Agosto de 2005.

Disponível em *http://shootout.alioth.debian.org/gp4/index.php*. Acesso em 20/12/2005.

BULTERMAN, D.; RUTLEDGE, L. **SMIL 2.0: Interactive Multimedia for Web and Mobile Devices**. Springer, Abril de 2004.

CENELEC. **Standardisation in digital interactive television**, Framework Directive 2002/21/EC, Abril de 2003. Disponível em *server.cenelec.org/NR/rdonlyres/ CCC0D181-330E-4275-BF5F-95DE06BA2298/0/DigitalTV_Sec0002_DC_Final.pdf*. Acesso em 28/08/2005.

COX, A. **Video4Linux Programming**. Maio de 2000. Disponível em *kernelbook.sourceforge.net/videobook.pdf*. Acesso em 20/12/2005.

DAVIC. **DAVIC specifications 1.5**, 1999. Disponível em: *www.davic.org/down1.htm*. Acesso em 02/02/2006.

DVB. **A Guide to Platform Harmonisation**. DVB White Paper, abril, 2004. Disponível em *www.dvb.org/documents/white-papers/wp05.platform%20harmonisation.final.pdf*. Acesso em 20/10/2005.

ECMA **Standardizing Information and Communication Systems. ECMAScript Language Specification**, Standard ECMA 262, 3rd Edition, 1999.

ETSI. Digital Video Broadcasting (DVB), **Multimedia Home Platform (MHP) Specification 1.1.1**, ETSI TS 102 812, Junho de 2003. Disponível em *www.mhp.org/*. Acesso em 28/08/2005.

FREED, N.; BORENSTEIN, N. **Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies**. IETF Request for Comments 2045, Novembro de 1996.

GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**, Eleventh Printing, Addison Wesley, 1995.

GEARY, D. M., MCCLELLAN, A. L. **Graphic Java: mastering the AWT**. SunSoft, Prentice Hall, 1997.

HAVi Organization, **HAVi-Home Audio/Video Interoperability**. 1999, Disponível em *www.havi.org/*. Acesso em 20/10/2005.

HORI, A., DEWA, Y. **Japanese Datacasting Conde Scheme BML**. IEEE Proceedings, Vol. 94, No 1, Janeiro de 2006.

HUNDT, A. **DirectFB Overview**. 2004, Disponível em *http://www.directfb.org*. Acesso em 23/06/2005.

IERUSALIMSCHY et al. **Lua 5.0 Reference Manual**. Technical Report MCC-14/03, PUC-Rio, 2003.

ISO/IEC 11172-1. **Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s - Part 1: Systems**. ISO Standard, 1993.

ISO/IEC 13522-4. **Information Techonology - Coding of multimedia and hypermedia information - Part 4: MHEG registration procedure**. ISO Standard, 1996.

ISO/IEC 13522-1. **Information Techonology - Coding of multimedia and hypermedia information - Part 1: MHEG object representation - Base notation (ASN.1)**. ISO Standard, 1997.

ISO/IEC 13522-3. **Information Techonology - Coding of multimedia and hypermedia information - Part 3: MHEG script interchange representation**. ISO Standard, 1997.

ISO/IEC 13522-5. **Information Techonology - Coding of multimedia and hypermedia information - Part 5: Support for base-level interactive applications**. ISO Standard, 1997.

ISO/IEC 13522-6. **Information Techonology - Coding of multimedia and hypermedia information - Part 6: Support for enhanced interactive applications**. ISO Standard, 1998.

ISO/IEC 13818-6. **Information technology - Generic coding of moving pictures and associated audio information - Part 6: Extensions for DSM-CC**. ISO Standard, 1998.

ISO/IEC 13818-1. **Information technology - Generic coding of moving pictures and associated audio information - Part 1: Systems**. ISO Standard, 2000.

ISO/IEC 15445. **Information technology – Document description and processing languages – HyperText Markup Language (HTML)**, maio de 2000.

ISO/IEC 8824-1. **Information Techonology -Abstract Syntax Notation One (ASN.1): Specification of basic notation**. ISO Standard, 2002.

ISO/IEC 14882. **Information technology - Programming languages - C++**, ISO Standard, 2003.

KUROSE, J. F., ROSS, K. W. **Redes de Computadores e a Internet: uma nova abordagem**, Pearson, Addison Wesley, 2003.

LAMADON, J.L. et al. **Usages of a SMIL Player in Digital Television. Proceeding Internet and Multimedia Systems and Applications**, Havaí, Agosto de 2003, p. 579-584.

Disponível em *http://www.lavid.ufpb.br*. Acesso em 20/12/2005.

MHP. **DVB-HTML Questions and Answers**, Abril de 2005. Disponível em *www.mhp.org/mhp_technology/mhp_1_1/dvbhtml/*. Acesso em 29/10/2005.

MORRIS, S., CHAIGNEAU, A. S. **Interactive TV Standards**. Focal Press, Elsevier, 2005.

MORENO, M. F. **Transporte de Dados IP sobre MPEG-2 e MPEG-4**, Monografia, Laboratório TeleMídia, Departamento de Informática, PUC-Rio, 2005.

MORENO, M. F. **Transporte de conteúdo MPEG-2 e MPEG-4 sobre redes IP**, Monografia, Laboratório TeleMídia, Departamento de Informática, PUC-Rio, 2005.

MORENO, M.F. et al. **Edição de Documentos Hipermídia em Tempo de Exibição**. XI Simpósio Brasileiro de Sistemas Multimídia e WEB – WebMedia, Poços de Caldas, Brasil, Dezembro de 2005.

MUCHALUAT-SAADE D.C. **Relações em Linguagens de Autoria Hipermídia: Aumentando Reuso e Expressividade da Linguagem NCL versão 2.0 para Autoria Declarativa de Documentos Hipermídia**, Tese de Doutorado, Departamento de Informática, PUC-Rio, Rio de Janeiro, Março de 2003.

NICHOLS, B. et al. **Pthreads Programming**, O'Reilly, 1996.

O'DRISCOLL G. **The Essential Guide to Digital Set-top Boxes and Interactive TV**, IMSC Press Multimedia Series, Prentice Hall PTR, 2000.

PEMBERTON, S. et al. **XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)**, 2002. Disponível em http://www.w3.org/TR/xhtml1/. 16/11/2005.

PÉREZ-LUQUE, M. J., LITTLE, T. D. C. **A Temporal Reference Framework for Multimedia Synchronization**, IEEE Journal on Selected Areas in Communications, 14, 1996.

PERKINS C. "RTP: Audio and Video for the Internet", Addison Wesley, 2003.

PERROT, P. **DVB-HTML – an optional declarative language within MHP 1.1**. Setembro de 2001. Disponível em *www.mhp.org/documents/mhp_perrot-dvb-html.pdf*. Acesso em 06/03/2006.

RODRIGUES R.F. **Formatação e Controle de Apresentações Hipermídia com Mecanismos de Adaptação Temporal**, Tese de Doutorado, Departamento de Informática, PUC-Rio, Rio de Janeiro, Março de 2003.

RODRIGUES R.F. et al. **Desenvolvimento e Integração de Ferramentas de Exibição em Sistemas de Apresentação Hipermídia**, VII Simpósio Brasileiro de Sistemas Multimídia e Hipermídia - SBMídia2001, Florianópolis, Santa Catarina - Outubro de 2001.

SCHWALB E. M. **iTV Handbook - Technologies and Standards**, IMSC Press Multimedia Series, Prentice Hall PTR, 2004.

SOARES, L.F.G. **Notas de Aula do Curso Fundamentos de Multimídia**, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brasil, Agosto de 2004.

SOARES, L.F.G., RODRIGUES, R.F., MUCHALUAT-SAADE, D.C. **Modelo de Contextos Aninhados – versão 3.0**, Relatório Técnico, Laboratório TeleMídia, Departamento de Informática, PUC-Rio, 2003.

SUN MICROSYSTEMS. **Java Media Framework API Specification**. 1999. *Disponível em* http://java.sun.com/jmf. Acesso em set. 05.

TANENBAUM, A.S. **Modern Operating Systems**, Prentice Hall, 1992.

UYTTERHOEVEN, G. **The Frame Buffer Device**. Maio de 2001. Disponível em *www.charmed.com/txt/framebuffer.txt*. Acesso em 29/10/2005.

W3C. **Cascading Style Sheets, level 2 - CSS 2 Specification**, W3C Recommendation, 1998.

W3C. **Document Object Model (DOM) Level 3 – Core Specification**. W3C Recommendation, 2004.

WHITAKER, J., BENSON, B. **Standard Handbook of Video and Television Engineering**. McGraw-Hill, 2003.

XFree. **The XFree86 Project**. 1994. Disponível em *www.xfree86.org/*. Acesso em 29/10/2005.

xine. **The xine Documentation**. 2006. Disponível em *www.xinehq.de*. Acesso em 27/03/2006.